Mathematical Formula Recognition and Automatic Detection and Translation of Algorithmic Components into Stochastic Petri Nets in Scientific Documents

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

By

ELISAVET ELLI KOSTALIA
B.Sc., Technical University of Crete, 2019

2021

Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

December 9th, 2021

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Elisavet Elli Kostalia ENTITLED Mathematical Formula Recognition and Automatic Detection and Translation of Algorithmic Components into Stochastic Petri Nets in Scientific Documents BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science

Nikolaos G. Bourbakis, Ph.D.
Thesis Director

Michael L. Raymer, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on
Final Examination

Nikolaos G. Bourbakis, Ph.D.

Soon M. Chung, Ph.D.

Euripides G.M Petrakis, Ph.D.
School of Electrical and Computer Engineering, Technical University of Crete

Barry Milligan, Ph.D.
Vice Provost for Academic Affairs
Dean of the Graduate School

WRIGHT STATE UNIVERSITY

ABSTRACT

Kostalia, Elisavet Elli, MSc, Department of Computer Science and Engineering, Wright State University, 2021. Mathematical Formula Recognition and Automatic Detection and Translation of Algorithmic Components into Stochastic Petri Nets in Scientific Documents.

A great percentage of documents in scientific and engineering disciplines include mathematical formulas and/or algorithms. Exploring the mathematical formulas in the technical documents, we focused on the mathematical operations associations, their syntactical correctness, and the association of these components into attributed graphs and Stochastic Petri Nets (SPN). We also introduce a formal language to generate mathematical formulas and evaluate their syntactical correctness. The main contribution of this work focuses on the automatic segmentation of mathematical documents for the parsing and analysis of detected algorithmic components. To achieve this, we present a synergy of methods, such as string parsing according to mathematical rules, Formal Language Modeling, optical analysis of technical documents in forms of images, structural analysis of text in images, and graph and Stochastic Petri Net mapping. Finally, for the

recognition of the algorithms, we enriched our rule based model with machine learning techniques to acquire better results.

Contents

List of figures

List of tables

List of images

Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Nikolaos Bourbakis, for his patience, guidance, and support. Thank you for giving me the opportunity to work with you, your wealth of knowledge, mentorship and caring have been immeasurably valuable for me. I appreciate and treasure everything you have taught me. I would also like to thank Dr. Soon Chung, as my committee member; your encouraging words and thoughtful, detailed feedback have been very important to me.

I owe a deep sense of gratitude to Dr. Euripides Petrakis for his constant support in both my undergraduate and graduate studies. Your keen interest to help your students grow has been immensely admirable, and your dedication, advice, and expertise have been invaluable to me. This journey would have been just a dream without your trust and guidance, I will remain forever grateful.

I feel extremely thankful for having in my life my parents, Nikoletta and Konstantinos, thank you for your endless support, you have always stood behind me, and this was no exception. Thank you for all the unconditional love. I would also like to thank my siblings, Michalis and Manolis, for their overwhelming support and inspiration. Last but not least, I would like to thank my friends, those thousands of miles away, and the ones closer, for their constant support and for believing in me. Thank you for always standing by me through the ups and downs of this journey, for your overpowering generosity, and your best suggestions.

1. Introduction

Algorithms and mathematical expressions are an integral part of computer science and the related literature. Documents in scientific and engineering disciplines present in a great percentage research findings and descriptions by introducing mathematical formulas or algorithms. Working towards the automatic understanding of the several components in documents, our purpose in this work is to contribute to the recognition and representation of how the mathematical formulas and the algorithmic components are structured and analyzed. Mathematical formulas are tightly connected to algorithms as not only algorithms usually contain the execution of several mathematical operations, but also, in many cases, algorithms are introduced in order to provide a step-by-step description of a certain mathematical formula. Here, our goal is to develop a methodology for the analysis of mathematical components found in technical documents and a system focusing on the detection and the extraction of algorithmic components in technical documents.

Exploring the mathematical formulas in the technical documents, we focused on the mathematical operations associations, their syntactical correctness, and the association of these components through their translation into attributed graphs and Stochastic Petri Nets (SPN) by processing the formulas in several layers. Initially, we conducted a first level comparative survey on the previous research works on parsing mathematical formulas in documents, which is presented in Chapter 2. The implementation of our system begins in Chapter 3 where we have developed a rule based methodology for parsing mathematical expressions, followed by mapping the symbol string representation of the mathematical formulas to an attributed graph and then to an SPN state machine in order to embed timing in the representation of the mathematical formulas. For a better understanding of the structure of a symbol string describing a mathematical formula, we designed a formal language which is introduced in Chapter 4. By making use of the formal grammar, we will be able to generate new mathematical formulas and evaluate their syntactical correctness.

Finally, as presented in Chapter 5, we designed and developed a system to automatically detect algorithmic components in documents and analyze them. We have implemented a rule-based methodology based on which, a document in a for of an image is segmented to image blocks. The image blocks after a pre-processing layer, are further analyzed to determine whether they contain algorithmic content. Next, we designed a model to convert the detected pseudo-algorithms in a graph, representing the sequence of steps introduced in the algorithm. Then, the detected algorithm is automatically mapped to an SPN state machine. The proposed algorithm analysis system makes use of a hybrid methodology of rule-based and machine learning procedures. Finally, in Chapter 6 we conclude this work and summarize the major findings and results. It also includes potential extensions of our methodology, where more complicated cases of mathematical formulas or algorithms will be taken under consideration.

2. Evaluating Methods for the Parsing and Recognizing of Mathematical Formulas in Technical Documents

2.1 Introduction

Scientific papers and other technical documents are usually composed by natural language text and other modalities, like block diagrams, mathematical formulas, tables, graphics, pictures, etc. The last two decades the Automatic Technical Documents Processing and Understanding (TDPU) has received more attention due to its profound applicability [1]. TDPU represents the continuation of the progress made in the fields of OCR, Natural Language Understanding, Pattern Recognition, and Image Understanding.

Surveys of research papers are usually divided into four different categories: brief surveys, descriptive surveys, first level comparative surveys and deep comparative evaluation. The first category includes a plain review of research methodologies informing the researchers on what papers are available in the field of study. The descriptive surveys refer to a deeper description of the approaches and their classification into various groups associated to certain characteristics, like bottom-up, top-down processing, etc. The first level comparative surveys approach offers a brief description of each methodology and then evaluates each of them by using a maturity function that illustrates the level of implementation and applicability.

Finally, the deep comparative evaluation of methodologies is based on a very thorough analysis of the performance of each method by running all of them on the same data set and providing details of their outcomes. This category, compared to the previous three, is the more unbiased approach because it is based on test results to evaluate the competing methodologies and to determine the most accurate, but at the same time is the most expensive and time-consuming [2].

One sub-area of TDPU is the recognition of mathematical formulas (MF). The MF area mainly deals with mathematical formulas detection in documents and the understanding process of formulas by using parsing methods. There are numerous research efforts in the field of mathematical formulas processing. For this effort here, about 200 papers were initially collected which, after preprocessing, were reduced to very small set by keeping those relevant to parsing. Thus, the purpose here is to conduct a comparative study among the finally selected papers by using a criterion of maturity. This criterion is defined based on a set of features associated with the importance for developing software methodologies for MF understanding. For instance, some of these features were complexity of the methodology, robustness, originality etc.

Segregating mathematical expressions have been grouped into two categories based on their position in the document: isolated and embedded. In this work here, we focus on isolated mathematical formulas in typeset documents. The goal here is to present an overview on this specific type of formulas, describing the parsing methods used during the structural analysis and interpretation of isolated MF. OCR, formulas detection and extraction are out of the scope of this effort. Here, we only evaluate the methods describing the syntactical parsing of the formulas. Thus, through each parsing method, the formula aims to be represented as an operator tree. In an operator tree structure, the internal nodes represent the operators, while the leaf nodes describe the operands. For the generation of the tree, the analysis layer may include several techniques that have been used so far, including rule-based and formal grammars.

There are several efforts studying the field of processing mathematical expressions. Thus, it is important to firstly report surveys studies associated to mathematical formulas. In particular, Chan and Yeung [3] presented a survey on both symbol recognition and structural analysis of mathematical expressions. They present various approaches developed on the parsing of the formulas to that date. Their work is mainly focused on the description of the similarities and the differences between the existing techniques. The survey by Zanibbi and Blostein [4] focuses on recognition methods of mathematical formulas. The unique contribution of that work is also the introduction to the study of mathematical formula retrieval area.

In both these efforts [3], [4] the emphasis is more towards to recognizing and understanding mathematical formulas however, understanding mathematical formulas involve parsing.

The rest of this thesis is organized as follows; in section II, the several approaches in parsing the mathematical formulas are presented. Section III presents an evaluation of state-of-the-art parsing methods highlighting the advantages and the limitations of each method through the evaluation process, using a maturity formula [5]. A number of features are selected for the evaluation of the maturity of each method, where each feature represents a different aspect of the evaluation. In section IV, the results of the evaluation are discussed, and future directions in mathematical formula analysis research are presented. Finally, section V states the conclusion of this work.

2.2 Recognizing and Representing Mathematical Formulas

The process of understanding mathematical formulas in documents is divided into four sub processes: (a) identification and segmentation which focus on detecting and isolating formulas in documents, (b) symbol recognition in formulas, (c) lay-out recognition for identifying the spatial relationships among symbols and, (d) content representation and analysis whose purpose to compute the outcome of the mathematical formulas. Parsing is included in the latter task, where the various objects (i.e. operators, operands) forming the formula are presented by an operator tree, which holds all the structural information of the mathematical expression. A large number of parsing techniques with a range of variations were introduced in the literature through the years for the analysis of mathematical formulas. The parsing is realized using either string grammars or two-dimensional grammars, depending on the system built.

The formal grammars used in the parsing process can follow either the top-down or the bottom-up approach. There are also cases where an integrated bottom-up and top-down approach is applied. The top-down approach is considered to be the fundamental structure processing technique. It processes the input structures starting with the global perspective of the input expression, and proceeds by analyzing horizontal and vertical relations among objects in the structure, which in our case are the sub-expressions in the mathematical formula. On the other hand, the bottom-up techniques process the elements in a mathematical structure by analyzing the nested structures based on specific objects (e.g. symbols, operators) within the structure.

A. Top-down parsing

The approach by Anderson [6] is one of the earliest works in this field. Despite its poor experimental results, the impact of this work on other works in the area of mathematical expression recognition is deemed significant. The work applies a top-down approach where, a syntax-directed algorithm, using rules of a formal grammar, is applied on the sub-expressions within the input formula. The experimental results exhibit the low efficiency of the method which may be attributed to the format of the formal grammar applied.

Chan and Yeung [7] introduce three mathematical expression parsing methods, namely, (a) symbol string parsing through backtracking, (b) parsing using binding symbol preprocessing and, (c) parsing using hierarchical decomposition. A Definite Clause Grammar (DCG) is executed within each method, and is implemented in a way that allows the parsing the mathematical formulas. DCG is highly declarative, which leaves no space for errors during the recognition process. It is executed by a Prolog interpreter (also used in the present work). The experimental results proved that, hierarchical decomposition is the most efficient method among the three in terms of parsing speed. In terms of complexity, the method aims to split expressions into smaller ones, so that even using a parser of high complexity, the time for parsing the short-length expression would be low. The method has been also applied for the understanding of handwritten mathematical expressions [8] with very high accuracy.

Tree structures are typical for describing the structural information of mathematical expressions. More specifically, binary trees have been used widely as they are both, easy to interpret and process and, capable of handling recursion. However, binary trees fail to represent all information in mathematical formulas especially in cases of mathematical formulas containing complex elements such as matrices, summations, integrals etc. The work by Toumit, Garcia-Salicetti and Emptoz [9] propose a flexible tree structure where each node may have more than two children nodes. A recursive method is applied initially on a single-node tree, containing the whole formula. While nodes can be complex objects, it recursively breaks each node into simpler object leaves based on the operators, comparators and spatial connectors in each object.

There can be a great deal of uncertainty in the interpretation of mathematical expressions mainly because of ambiguity inherent in mathematical notation and this might make interpretation dependent on human experience. In order to deal with this problem, Chen, Shimizu and Okada [10] introduce a rule-based approach for the automatic parsing of mathematical expressions. They first extract a layout tree along with a semantic tree representing the layout and semantics of mathematical expressions respectively, prior to applying a set of mathematical, sense-based and experience-based rules. In an extension of this work [11],, the authors discuss the various ambiguity issues in mathematical expression understanding.

In the work by Jin, Han and Wang [12], mathematical formulas are parsed by applying a hierarchical and recursive decomposition process that computes an operator tree as a result. Processing is split into three layers; each layer is dedicated to different mathematical elements represented by glyphs. During the first layer, the most basic elements are processed, like fractions, radicals, and delimiters, which outputs the compound expressions of the formula. A multi-line mathematical formula is then transformed into a one-dimensional array. The processing of this array is based on the backbone glyph extraction. Going towards the next layers, each compound expression is handled as an individual glyph, representing a subexpression. The process terminates when there is no more subexpression for the formula to be split.

Toyota, Uchida and Suzuki [13] handle the parsing of a mathematical formula as an OCR verification step that applies a context-free grammar capable of dealing with mathematical syntax. This is a top-down approach where the grammar is applied on the tree representation of the formula and grammar rules are defined according to Anderson [6].

B. Bottom-up parsing

Lavirotte and Pottier [14] introduce a graph grammar approach. The input formula is represented by a graph structure which is generated based on the spatial locations of the symbols in it. Graph nodes represent symbols in the formula and graph edges represent their relative positions. The graph is then transformed to a syntax tree using a graph grammar. The graph grammar is a context-sensitive graph grammar where, the terminal symbols represent mathematical symbols and, nonterminal symbols represent mathematical expressions. The challenge in graph building relates with the number of links: a very big or small number of links might lead to ambiguities: might lead to more than one formula or, might not be able to represent all information needed for building a formula, respectively. Along the same lines and following the optical recognition of symbols in the formulas, Chaudhuri and Garain [15], [16] extract the logical relationships among the formula components. The process is based on the idea of building the layout of a formula using the spatial relationships of its components and their bounding box coordinates as parameters. For the final part of the syntactic parsing and mathematical formula understanding, a number of pre-defined rules are applied.

In the work by Guo, Huang, Liu and Jiang [17], a mathematical expression is decomposed into sub-expressions. The method introduces the idea of continuous reformation of the global expression structure by decomposing the formula into basic sub-expressions, and by appending the analysis results to the higher levels of the bottom-up process. The script relation trees are generated by applying a context free grammar and an N-best algorithm for the finishing analysis tasks.

C. Integrated top-down and bottom-up parsing

Integrated parsing approaches combine both top-down and bottom-up parsing techniques. Fateman et al. in [18], [19] handle typeset mathematical expressions using both OCR and structural analysis. Structural analysis applies a bottom-up parser. Prior to structural analysis, a top-down method is applied for identifying and parsing sub-structures in a formula. The experimental results for the bottom-up method are not promising due to the complexity of the inputs. This result, however, does not rule-out the use of the bottom-up approach on other inputs.

In DRACULAE system, Zanibbi and Blostein [20] process a mathematical formula left-to-right. The so called Baseline Structure Tree (BST) is generated first and is transformed to a Lexed BST [21]. This is then translated to a LaTeX expression which is forwarded to the expression analysis stage which produces the operator tree. Each expression is analyzed in terms of syntax and semantics. During syntax analysis, a context-free grammar is applied on the linearized symbol string and a parse tree is produced. During semantic analysis, a set of tree transformation rules are applied for detecting implicit operations and the operator tree is re-ordered. The complexity of DRACULAE is linear on the average.

Takiguchy, Okada and Miyake [22], apply both a layout and a semantic tree for the understanding of a mathematical formula and its translation to LaTeX. Following the layout analysis of a formula, Guo et al. [17] use sense-based and experience-based rules.

2.3 Evaluation

In the following, we present a comparative evaluation of the methods referred to above. All methods are evaluated using the features of Table 1. These features are deemed representative of their operation, purposed and expected result [23].

Features	Description
Reliability (F1)	The methodology produces expected results under normal operating conditions
Robustness (F2)	Results are produced under extreme conditions – formulas with a great complexity
Complexity (F3)	The difficulty in implementing a methodology due to a large number of components or associations. Also refers to Computational and Memory requirements.

Efficiency (F4)	The methodology can achieve the desired results in an efficient way
Originality (F5)	A novel methodology is presented
Accuracy (F6)	The precision of the results
Speed (F7)	Processing time of the methodology presented
Experiments (F8)	Size of experimental data
Further Improvements (F9)	Enhancements required in the design
Cost (F10)	The Implementation cost of the methodology
Portability (F11)	The ability of the system to work in different platforms
Parsing Method (F12)	The parsing method used for the syntactical analysis of the formula TD for top-down approaches, BU for bottom-up and IN for integrated approaches

Table 1 - Evaluation characteristics

The selected features are defined based on inherent characteristics (i.e., implementation complexity, accuracy, extensibility, originality, robustness) or, the performance (i.e. efficiency and quality of the results) of the parsing methods which they are related with. In order to achieve a more quantitative assessment, all competing methods are rated based on two perspectives, one associated with the end-user and the other associated with the developer. The weights w_j for the developer and user perspective are shown in Table 2. They are defined by our evaluators, and they are used for computing a "maturity score" M_i for each methodology.

The maturity function of Eq. (1) defines the maturity score for a method taking into consideration the weights of each perspective.

$$M_i = \frac{\sum_{j=1}^{N} w_j f_{ij}}{\sum_{j=1}^{N} w_j} \qquad (1)$$

In Eq. (1), N represents the number of quantitative features used for the evaluation (i.e. 11 in our case). Each methodology is assigned a score from 1 to 5 for each feature. A score 1 denotes poor performance of the approach on feature, while a score 5 denotes very good performance, respectively. The features showcasing the further improvements (F9) and the cost (F10), reflect a negative impact as they describe the required enhancements for the specific methodology and the implementation cost of the system by incorporating the respective feature. For both these negative impact features, a higher score denotes the less requirements for the maximum performance. The scores assigned to each methodology, based on the proposed features are shown in Table 3.

Each methodology receives a score f_{ij} for each feature (means each methodology receives 11 scores). Following this, for the computation of the maturity of a methodology for a perspective (i.e. end-user or developer) the weight of each feature from the developer's perspective is multiplied with each distinct feature score. The summation of all these products is normalized by the summation of the weights for this specific perspective.

Features	Weights	
	End-user (W_U)	Developer (W_D)
F1	1	1
F2	1	1
F3	0.1	1
F4	1	0.8
F5	0.1	0.9
F6	1	1
F7	1	1
F8	0.4	1
F9	0.5	0.8
F10	0.6	0.9
F11	0.3	0.9

Table 2 - Weights assigned to features, for the end-user and developer perspectives

The result of the division will form the maturity value for the given methodology for the developer's perspective. For example, for the method by Fateman and Tokuyasu [18] for the end-user's perspective is:

$$M_{[15]} = \frac{3\times1+2\times1+3\times0.1+3\times1+3\times0.1+2\times1+3\times1+1\times0.4+1\times0.5+5\times0.6+3\times0.3}{1+1+0.1+1+0.1+1+1+0.4+0.5+0.6+0.3} = 3.6$$

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
[18]	3	2	3	3	3	2	3	1	1	5	3	IN
[7]	3	2	3	4	5	3	4	1	1	3	1	TD
[14]	3	2	4	4	5	2	4	1	2	2	3	BU
[9]	4	2	3	4	4	3	4	1	2	4	3	TD
[10]	3	2	2	3	5	3	2	3	2	3	2	TD
[15]	5	4	3	3	4	5	3	3	4	2	4	BU
[8]	5	4	5	5	3	5	4	4	4	1	4	TD
[12]	5	4	4	4	2	5	3	4	3	1	4	TD
[20]	4	4	4	3	2	5	3	3	4	2	4	IN
[22]	4	2	4	2	3	4	2	2	2	4	5	IN
[13]	4	3	2	4	4	3	1	3	3	2	2	TD
[17]	4	3	3	3	2	3	2	5	3	2	3	BU

Table 3 - End-user and developer scores for all methods.

2.4 Discussion on Evaluation Results

The overall maturity scores are calculated based on the Eq. (1), for both user and developer perspectives. The formula indicates how mature each individual methodology is, the time it was developed and not in comparison to each other. Figure 1 illustrates the maturity scores for each methodology, based on both perspectives. Figure 2 illustrates the average maturity score for each methodology. Figure 2 shows the scores without taking weights into consideration. No method reached the maximum maturity score. However, most methods achieve relatively high scores. The method by Chan and Yeung [8], which applies top-down parsing, outperforms all other methods achieving average score 4.10/5.00, followed by the method by Chen, Shimizu and Okada [11] which also applies top-down parsing and achieves average score 3.70/5.00. Third in order, in the integrated method by Zanibbi and Bolstein [20] which reached 3.54/5.00 average score.

All parsing methods proved successful on mathematical formula understanding although the methods differ from each other in terms architecture used, nature of the system or application within they are applied and other factors. Therefore, the decision of which method is the preferred one resorts to the end-user or developer who needs to take all these factors into account.

The top-down parsing approaches with the best scores are [8] and [12]. The first one relies on backtracking which does not guarantee very good efficiency in the general case [23]. The second one makes relies on recursion for the decomposition of input formulas which might result is lower computation cost. Furthermore, the integrated methodology Zanibbi, Blostein and Cordy [20] has also achieved low computational cost due to its linear time complexity in the average case. This improves the maturity score.

The integrated approaches are expected to be very prominent. This hypothesis is based on the understanding that top-down approaches resemble the way the human brain understands the components of a formula in the first place. Also, the bottom-up approach for the evaluation of the subexpressions of the formula leads to the evaluation of the complete mathematical formula. The evaluation results are not according to this hypothesis. Notice that, lower maturity are obtained for methodologies which have been implemented to their full extend (i.e. as full-fledged systems supporting all stages of recognition and understanding) which can be both, very complex and computationally expensive.

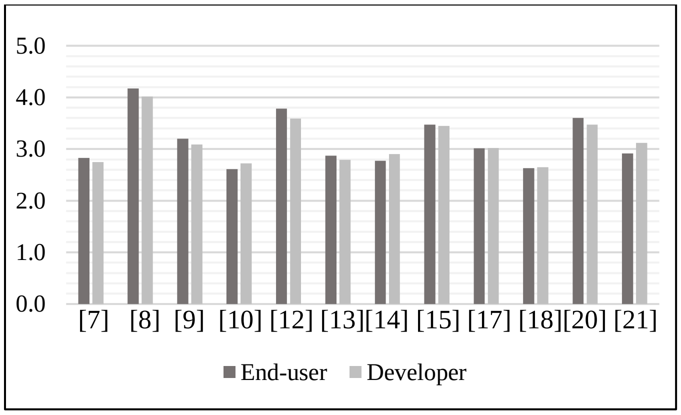


Figure 1 - Maturity scores according to end-user and developer perspectives.

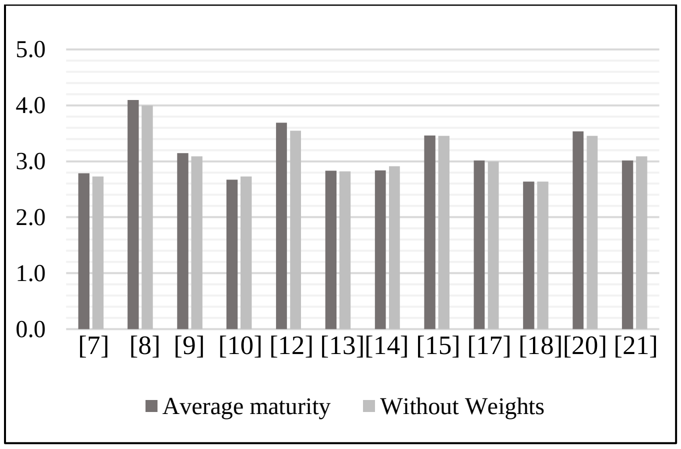


Figure 2 - Average maturity scores with and without weights

2.5 Conclusion

We present a comparative study on mathematical expression and formula understanding methods. The discussion and following evaluation is based on a number of criteria relating with the operation of each method and whose purpose is to reveal the strengths and weaknesses of each method. A maturity metric is also introduced which becomes the overall criterion for ranking the competing parsing methods by efficiency taking also into consideration criteria pertinent to end-users or system developers.

All comparisons are made between the works themselves (to their full potential) and not among the different systems within which the methods are applied. By conducting this survey, we concluded that no approach could achieve the maximum maturity in the field of understanding mathematical formulas. This can become possible by developing methods simulating the hierarchy of operations of human mind or, by using structures that would be able to hold, not only the structural, but also the functional information of a formula. We have examined the process of formula parsing and how close this is to the way the human-brain processes the formulas. It is a step closer to machine deep understanding of technical documents that may be used to train machines handle mathematics [24], [25].

3. Conversion of Mathematical Formulas into Graphs and Stochastic Petri Nets

3.1 Introduction

Even though mathematical expressions consist of well-defined rules applied on the syntax and the operations forming the hierarchy of the operations, the analysis of the components and their associations tends to be a challenging part of any machine trying to analyze and understand this expression. Since understanding of mathematical expressions has a certain connection with people's sense and experience, we build a system which takes under consideration the mathematical rules in addition to the rules based on the human sense and experience to understand expressions perfectly and to avoid problems of uncertainty.

A rule base approach is set up in this work which consists of mathematical, sense-based and experience-based rules to help us understand the expressions correctly and naturally. The mathematical rules are helpful to automatically and unambiguously parse the structure and the semantics of an expression after having recognized characters and obtained information for the spatial relationship of the operators in a tree structured format. While the sense-based rules provide the handling of the expression’s ambiguousness in layout, the experience-based rules are responsible for dealing with uncertainty in expression semantics. The final purpose of this part is to design a system which takes as an input a mathematical expression and generates the Stochastic Petri Net representation of this expression.

Conventionally, we use as a source a mathematical formula in LaTeX format which comes from a free OCR tool which converts an image of a mathematical formula to LaTeX code. Thus, the only assumption used here is a preprocessing of inserting an image to an external OCR tool and receiving an MathML or TeX (LaTeX) formatted output as the input to our system. This is quite realistic as almost all the methods proposed in the literature give recognition output in one of these formats.

3.2 Mathematical expressions in technical documents

Technical documents include, among other modalities, mathematical expressions, which may be found at a great percentage of documents, especially in the area of Computer Science. This work is exclusively focused on Mathematical expressions found in technical documents. In the very beginning it is essential to define what a mathematical expression in a technical document is: it is a finite combination of symbols that is well-formed according to rules that depend on the context. Mathematical symbols can designate numbers (constants), variables, operations, functions, brackets, punctuation, and grouping to help determine the order of the operations and other aspects of logical syntax.

An expression is a syntactic construct which must be well-formed. The operators in the expression must have the correct number of attributes in the correct places. Any string of symbols which violates the rules of syntax is not considered well-formed and is not identified as a valid mathematical expression. For example, the expression $1+2\times3$ is well-formed, but the expression $9\times4)x+/y$ is not.

3.3 Conversion from electronic type of the technical document (pdf) to the SPN representation

Our proposed methodology consists of five sequential layers of processing; a document in a pdf format shall form the input, which after the detection of the mathematical parts will be split in individual images which will then undergo Optical Character Recognition and result to a symbol string in a LaTeX format. The recognized symbol string will then go through parsing and following, will get translated to an attribute graph, and to an SPN representation. In this part of the work, we present the processing layers starting after the OCR of an image and moving towards the generation of the Stochastic Petri Net. Although the OCR process has not been implemented at this time, a brief description of the optical recognition process would be valuable to be introduced.

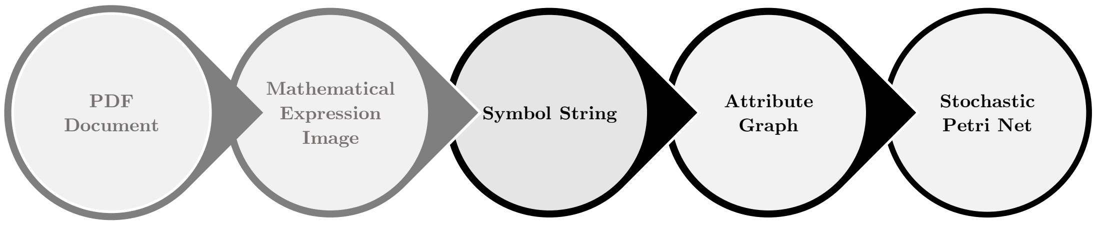


Figure 3 - Steps from initial form of technical document to SPN representation of mathematical expression.

3.4 Optical Character Recognition OCR

Mathematical expression recognition involves three major tasks: segmentation - detecting symbols, classification, and parsing - determining expression structure. These tasks may be solved in a sequential feed-forward manner, or in a globally integrated fashion.

- Segmentation

It is a task of grouping related primitives. These primitives could be pixels from an image, or strokes from a handwritten equation. The main challenge of symbol segmentation in typeset mathematical expression images is fractured symbols whose components were split by printing and scanning noise [26].

- Classification

Common algorithms for symbol classification include nearest neighbor, support vector machines, random forests, hidden Markov models, convolutional neural networks, and bidirectional long short-term memory networks.

- Parsing

Converting input primitives (e.g., images, handwritten strokes, or symbols) to a description of formula structure. A common set of features used to represent the spatial relations between components are geometric features.

The use of the relative position of the symbols gives additional information about the association between symbols and elements of the expression, having as an example the superscripts and subscripts or operators such as summation and integrals.

Reading Mathematical Expressions is executed in a left-to-right order, following the precedence of operations. The precedence of operations is a collection of rules that reflect conventions about which procedures to perform first in order to evaluate a given mathematical expression and its purpose is to eliminate ambiguity while interpreting an expression. It is also a way that allows notation to be as brief as possible. The simplest way of parsing a two-dimensional expression is to translate it into its equivalent one-dimensional representation and then parse it using an existing parser. An expression consists of one or more mathematically linked symbol groups. A symbol group is defined to be a special mathematical symbol which may deviate from the typographical center of a mathematical expression and the symbols that appear with it [27], [28]. For example, $\sum_{i=1}^{n} x_i$ is a symbol group. When all symbol groups in an expression have been grouped, the expression can be transformed from a 2-D form to 1-D, according to the previously mentioned OCR steps. In this work we did not focus on the Optical Character Recognition of the mathematical formulas, we rather take the output of the OCR tool in the form of a symbol string as a given. We will handle this issue in the future.

3.5 How we process the mathematical expressions

Given a string expression at the input, and before it is split into its left and right parts, an equation detection procedure is applied in order to detect whether this is an equality or inequality type of expression and then it is split to the two parts of it: left and right part. The two parts are treated as distinct mathematical expressions and are processed individually and in a next processing layer we will handle the connection of the two parts. For each part, the parentheses inside the mathematical expression are detected and processed -as they are in the highest priority in the hierarchy- as individual expressions as they may include nested parentheses. In general, we find different types of expressions and calculate the simple or complex expressions inside them. We follow the precedence of the operations inside every sub-expression which is a simple expression, then move to the outer layer of hierarchy. Our model goes through different levels of hierarchy, depending on the context of the expression. At each level, the elements of the current operation hierarchy level are processed, and the outcome of this process will replace this sub-expression, modifying the input expression. Each mathematical operation is given an id number for identification purposes, for example, the first summation operation is marked as add1 and the first detected parenthesis is marked as parnethesis1. An example of this procedure is shown in the following image.

```
Input: ['y', '=', 'e', '*', 'c', '/', '6', '+', '(', 'a', '-', 'b', ')']
Part1: ['y']
Part2: ['e', '*', 'c', '/', '6', '+', '(', 'a', '-', 'b', ')']
Part2-Step1: ['e', '*', 'c', '/', '6', '+', 'parenthesis1']
Part2-Step2: ['mul1', '/', '6', '+', 'parenthesis1']
Part2-Step3: ['div1', '+', 'parenthesis1']
Part2-Step4: ['add1']
```

Figure 4 - Proceeding steps of the expression
y=e*c/6+(a-b)

3.6 The mathematical expressions

1.1 We define as simple expressions the expressions that include numbers and alphabet letters along with the basic operators +, -, *, / and process them in the order multiplications and divisions, then additions and subtractions.

1.2 Types of Mathematical Expressions Modules which our system processes:

- Parenthesis
- Equality - Inequality
- Simple expression
- Fraction
- Summation
- Finite Integral

- Factorial
- Root
- Exponential
- Limit
- Indefinite Integral
- Absolute Value
- Logarithmic

3.7 Math Expressions to Graphs

Each mathematical expression is represented by a graph. Each sub-expression becomes a sub-graph of the full graph. That means that for every level of the hierarchy and for each operation, two nodes and two edges are created. The first node that is created is the result of the operation that will be registered as a new node along with the edge that will connect the node that represents the last operation's result. The second node is the next operand to be executed which will be associated, through a new edge, with the result of the current operation.

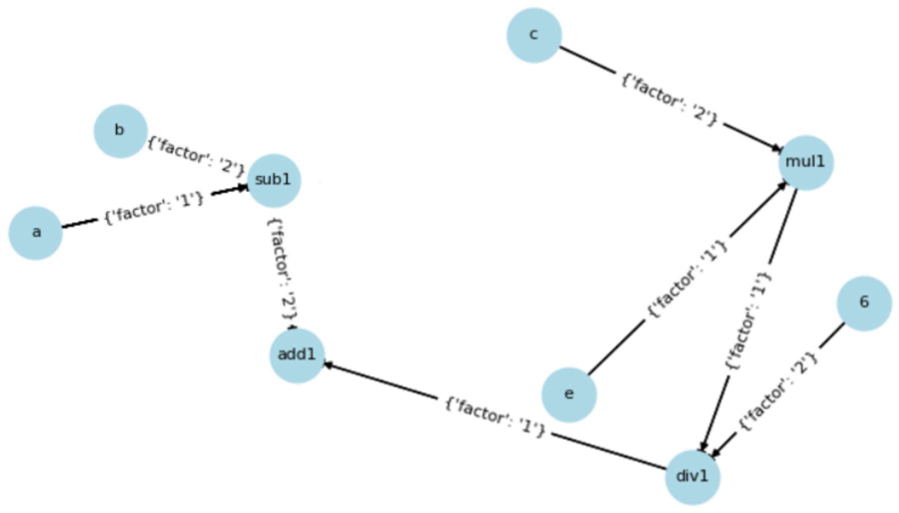


Figure 5 - The attributed graph of the expression
e*c/6+(a-b)

For each operator, there is a number of operators that are required in order for the operation to be executed. For example, for simple operators such as addition and subtraction two operators are required, though for a fraction, we are expected to have two mathematical expressions, one as the numerator and one as the denominator. The attributes on the edges connecting two nodes in the graph represent the type of the operator that is inside the node that participates in the operation.

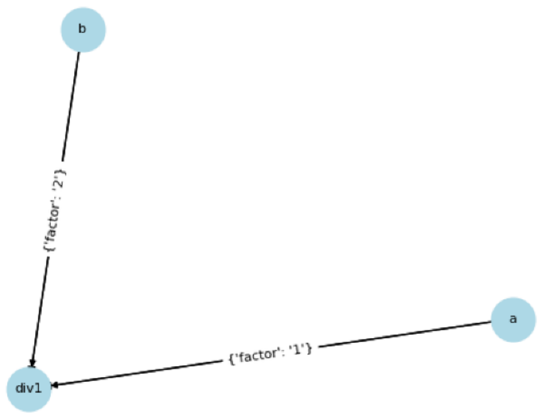


Figure 6 - The graph generated representing the expression a/b. Node 'a' constitutes the first operator-numerator of the division and node 'b' constitutes the second operator of the division-denominator. Both edges end up to the node 'div1' which is a keyword representing the first (and only, in this specific example) division operation.

Operation	**Number of operands**	**Names of operands**	**Syntax**
Addition	2	factor1, factor2	factor1 + factor2
Subtraction	2	factor1, factor2	factor1 - factor2
Multiplication	2	factor1, factor2	factor1 * factor2
Division	2	factor1, factor2	factor1 / factor2
Fraction	2	numerator, denominator	frac{numerator}{denominator}
Factorial	1	factor1	factor1! *or* (factor1)!
Exponential	2	base, exponent	base^exponent *or* (base)^{exponent} *or* (base)^exponent *or* base^{exponent}
Integral	4	lower limit, upper limit, function, differential	int{lower limit}{upper limit}functiond{differential}
Summation	3	factor1, factor2, function	sum{factor1}{factor2}{function}
Root	2	factor1, function	sqrt[factor1]{function}
Absolute Value	1	factor1	|factor1|
Logarithmic	2	factor1, factor2	log {factor1}{factor2}
Limit	3	factor1, factor2, function	lim{factor1}{factor2}{function}

Table 4 - Different types of expressions, the operands required and their syntax.

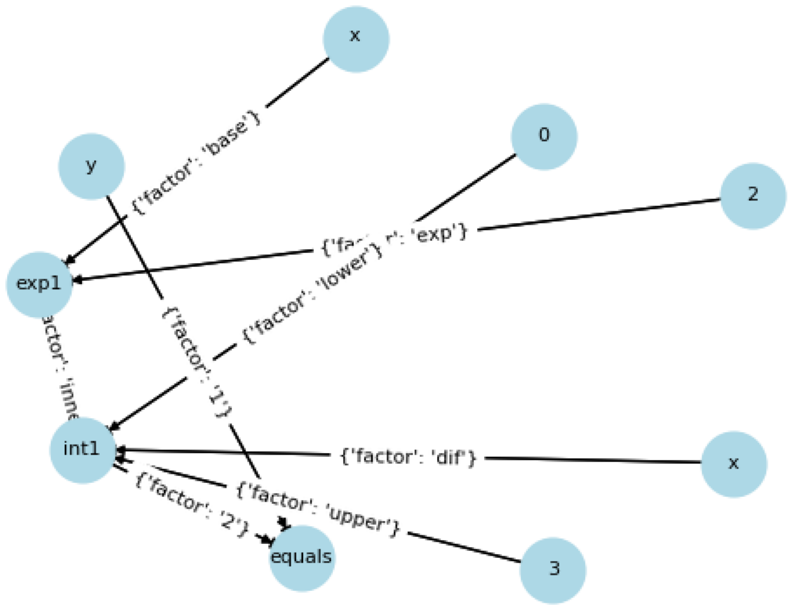


Figure 7 - The graph generated representing the expression y=int{0}{3}k^2d{x}.

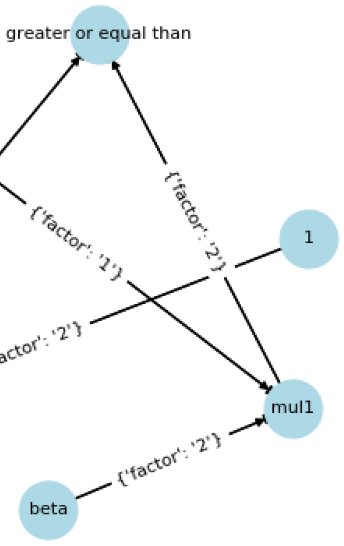


Figure 8 - The graph generated representing the expression y-1>=alpha*beta.

3.8 Mathematical Formulas into Stochastic Petri Nets

Stochastic Petri Nets are used to describe and analyze systems that are concurrent, distributed parallel and non-deterministic. They provide functional information and are also used as a machine language for development, simulation, and applications. Petri net is an information flow model which we are using in order to interpret the mathematical expressions along with their functionality. A Petri net is a directed bipartite graph in which the nodes represent transitions (i.e. events that may occur) represented by bar, and places (i.e. conditions), represented by circles. Compared to graphs which provide with structural information alone, SPNs provide also with functional information (i.e. timing and synchronization) of the operations inside a mathematical expression

3.8.1 Stochastic Petri Nets prerequisites

In this section we provide the basic Stochastic Petri Net (SPN) prerequisites. SPN is a specialized category of Petri Nets thus, SPNs and Petri Nets have the same visualization components and go by the same visualization rules. We will demonstrate how a basic component of Petri Nets is represented visually. A Petri net consists of places, transitions, and arcs [29], [30], where arcs can connect a place to a transition or vice versa, but an arc can never connect two places or two transitions. Places in a Petri net may contain a discrete number of tokens. Arcs are characterized by their capacity, which is the number of tokens they are able to transfer. Any distribution of tokens over the places will represent a configuration of the net called a marking. In our mapping we use the default capacity of 1. A transition of a Petri net is enabled when there are sufficient tokens in all its input places, which means that the number of tokens in each of its input places is at least equal to the arc weight going from the place to the transition. A transition may fire if it is enabled. When the transition fires, it consumes the required input tokens, and creates tokens in its output places. This results in a new marking of the net, a state description of all places. In a graphic representation of a Petri net in Figure 9, places are depicted with circles (where each circle contains or not one or more dots called tokens), transitions with long narrow rectangles, and arcs as one-way arrows that show connections of places to transitions or transitions to places. Labels above arcs indicate their capacity, which means the maximum number of tokens that an arc can carry simultaneously [31]. An inhibitor arc is represented by an arc terminated with a small empty circle [32]. More information about Petri Nets and Stochastic Petri Nets can be found in the corresponding literature [29], [30], [32].

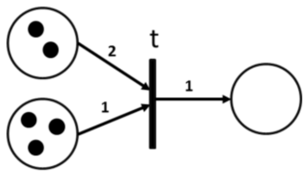


Figure 9 - A simple Petri Net graph

Here, we present some basic points for the representation of mathematical expressions, operands, and operator's results into SPN graphs. So, simple mathematical operations or functions are represented by timed transitions (thick rectangles), since they are the transitionary layer between the variables/operands and the result of the mathematical operation/function. Places (big circles) represent variables or constants that describe any operand as a part of a mathematical operation. Places may also represent the result of an arithmetic operation/function execution which will eventually become an input to a forthcoming operation.

3.8.2 Methodology We Follow to Convert Graphs to SPN

Having as the final purpose of this part of the work the conversion of a mathematical formula to a Stochastic Petri Net, the graph was designed to be generated in such way so the variables and the numbers of the expression can be viewed as the places of the Petri Net and the operations between these operands as the transitions. Using the generated graph of the mathematical expression as the input, for each node that represents an operation, a transition is created which describes the operation being executed, given the attributes of the two incoming edges. Each operand of an arithmetic operation or function in the expression (variable or constant or result of an operation's execution) is represented by a place. The attributes of edges leading to the same node describe the several operands required for an operation to execute so any edge of the generated graph will point to a transition. This transition represents the arithmetic operation/function which will be executed using the attributes of the incoming edges. Finally, a new place will be created which will represent the result of the operation's execution and an arrow will be connecting the transition with the new place. This process will continue until all nodes of the graph have a corresponding place in the SPN graph. For the creation and representation of the Petri net the SNAKES library of Python was used [33].

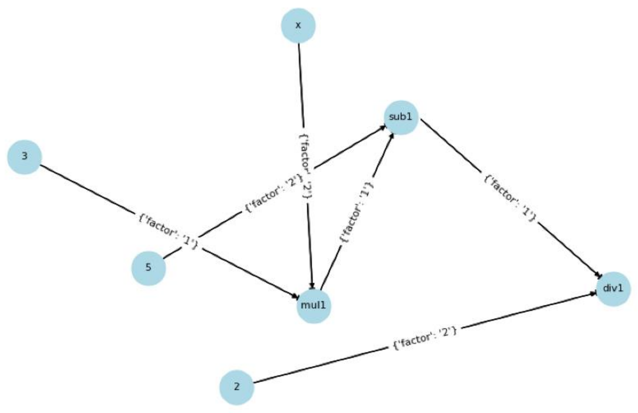


Figure 10 - Graph representation of the expression (3x-5)/2

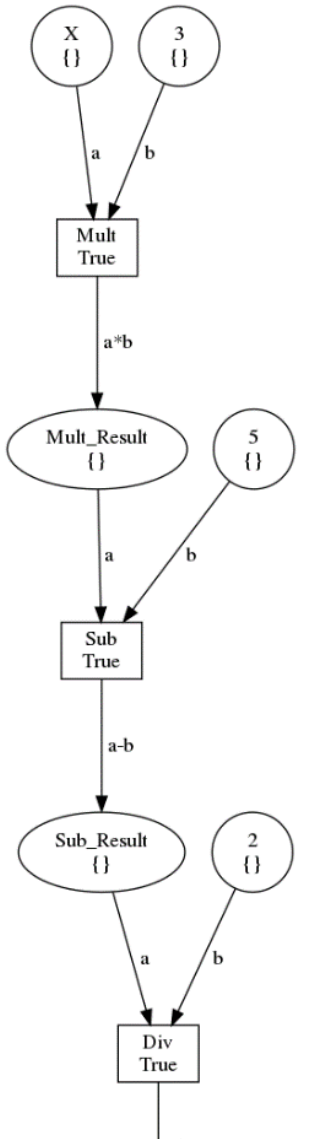


Figure 11 - SPN representation of the expression (3x-5)/2

The symbol string is being parsed based on the alphabet letters and the operators, identifying numbers that are made up with more than a numerical symbol and variables and operator key-words consisting of more than one letters. The method also detects implicit multiplications between elements of the mathematical expression. Implicit multiplications are expressed by two symbols one next to the other without an operator between them, where a multiplication operation is implied. Given the fact that the expression is formed by two parts and an equality or comparative operator, the expression is being split into two parts. These two parts are processed individually using the exact same procedure. While parentheses have highest priority in any mathematical expression, parentheses are being detected together with expressions within parentheses form sub-expressions which are being isolated and processed separately based on the precedence of operations. While executing the operation of each sub-expression, we keep in memory the index of the previous node so that we create one graph of all the sub-expressions. The graph we are creating will be a graph in a form of a binary tree - each parent node will have maximum two child nodes. Following this step, based on the precedence of the operations mentioned above, each sub-expression is being transformed to nodes and edges, creating an attributed directed digraph that describes the relationships among the different components of the initial expression.

In the next level of processing the mathematical expression, we detect different types of operations such as fractions, exponentials, roots, summation, integral, factorial, absolute values, that also include sub-expressions and process them just like the expressions in parentheses. Every new input of the generated graph consists of three elements: the starting node, the ending node and the attribute of the edge that represents the factor in the syntax of the operation. After finishing the process of both parts of the input expression, the value of each one is assigned in a new node with an edge that points towards the keyword representing the relation between these two parts. When all the operands and operators are included in the graph, we use the NetworkX library of Python so we can visualize it. Final step is to create the SPN representation based on the methodology described above. The following diagram shows the sequential steps that are followed to achieve the SPN representation of the mathematical expression through the graph.

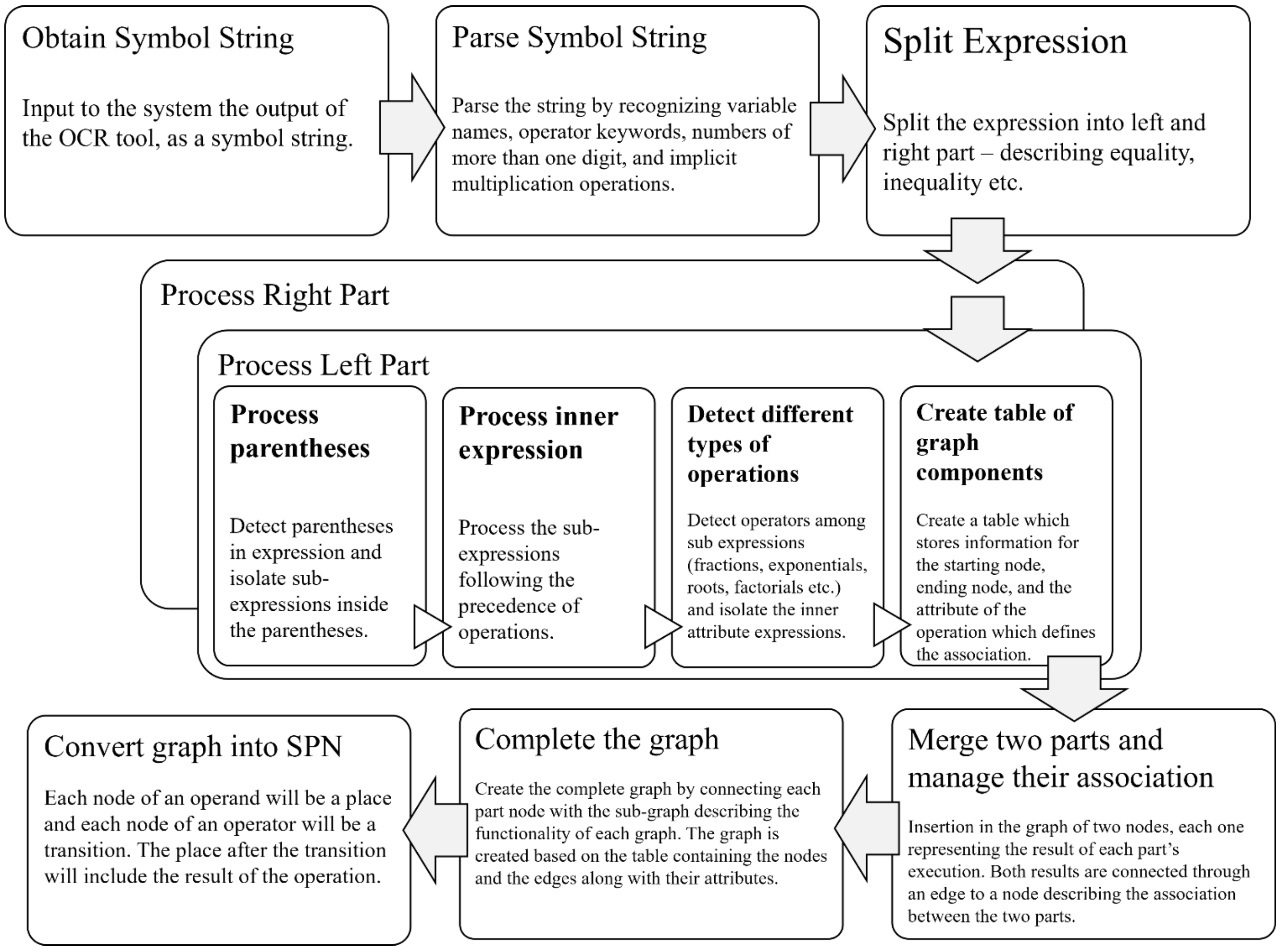

Obtain Symbol String
Input to the system the output of the OCR tool, as a symbol string.
Parse Symbol String
Parse the string by recognizing variable names, operator keywords, numbers of more than one digit, and implicit multiplication operations.
Split Expression
Split the expression into left and right part – describing equality, inequality etc.
Process Right Part
Process Left Part
Process parentheses
Detect parentheses in expression and isolate sub-expressions inside the parentheses.
Process inner expression
Process the sub-expressions following the precedence of operations.
Detect different types of operations
Detect operators among sub expressions (fractions, exponentials, roots, factorials etc.) and isolate the inner attribute expressions.
Create table of graph components
Create a table which stores information for the starting node, ending node, and the attribute of the operation which defines the association.
Merge two parts and manage their association
Insertion in the graph of two nodes, each one representing the result of each part's execution. Both results are connected through an edge to a node describing the association between the two parts.
Complete the graph
Create the complete graph by connecting each part node with the sub-graph describing the functionality of each graph. The graph is created based on the table containing the nodes and the edges along with their attributes.
Convert graph into SPN
Each node of an operand will be a place and each node of an operator will be a transition. The place after the transition will include the result of the operation.

Figure 12 - Data Flow Diagram describing the procedure of converting a symbol string mathematical expression to Stochastic Petri Net representation.

4. Generation of Mathematical Formulas using a Formal Grammar

4.1 Introduction

Mathematical formulas consist of combinations of different mathematical expressions, the associations of which, are accurately defined using the well-known mathematical notation. A mathematical expression requires that both, the operators, and the operands within the expression are defined. The complete process of the generation of MF constitutes of three layers of processing, representing the steps that are strongly connected and essential for the interpretation of a formula:

1. Generation of a MF using the syntax of the formal grammar designed
2. Syntactical Optimization - Elimination of undefined terms inside the formula
3. Semantical analysis and interpretation of the formula

In this phase of our work, we have implemented steps 1 and 2. During the first layer of processing, the formal grammar of our language is executed, and mathematical formulas are generated. In the second step, these generated formulas go through filters to evaluate their syntactical content, in a way that undefined terms will be eliminated, or excess notation will be removed without changing the syntax of the formula.

While mathematical expressions are typically illustrated as two-dimensional structures of math symbols in either handwritten form or images, meaning that each mathematical symbol obtains a relative positions to another in the 2-

dimensional space in an image, a high-quality typesetting system was developed for the description of mathematical notation in scientific documentation: $\LaTeX$ is widely used in typesetting of complex mathematical formulas and is established as a standard for the communication and publication of scientific documents [34]. In these terms, in our work, we follow the $\LaTeX$ format to describe each mathematical element or mathematical formula. To acquire the syntactical correctness of the mathematical language, a formal language (FL) is designed. By making use of this FL and its formal grammar subsequently, we make sure that the MF are formed based on specific grammar rules which define their syntactical correctness and validity [35].

4.2 Mathematical Operations

A representative number of mathematical expressions found in published scholar and technical documents was collected and studied, and the most frequently used mathematical operations (functions) were congregated in a list, illustrated in Table 5. They form the corpus of the different operations that will be used to construct a new formula, and each operation corresponds to a distinct letter of the alphabet of the introduced language. Each distinct mathematical operation is handled uniquely, based on the syntax of the code according to the LaTeX format and the number of operands that are required to define it completely. For a more convenient processing of the symbol strings describing the formula, we eliminate special symbols used to describe the spatial relations of elements in the expression (like "\", " ^ ", and "_") from the symbol string of LaTeX format.

Serial number	Operation	Operation Name	Number of Operands	Pre-processed LaTeX format	LaTeX format
1	a+b	Addition	2	a + b	a + b
2	a-b	Subtraction	2	a - b	a - b
3	$a * b$ $a \cdot b$ $a \times b$	Product	2	a * b	a * b a \cdot b a \times b
4	a/b	Division	2	a / b	a / b
5	$\frac{a}{b}$	Fraction	2	frac{a}{b}	frac{a}{b}
6	a!	Factorial	1	a!	a!
7	a^b	Exponential	2	a^{b}	a^{b}
8	$\int a\, dx$	Indefinite Integral	1	int{a}d{x}	\int{a}d{x}

9	$\int_{a}^{b} c\,dx$	Definite Integral	4	int{a}{b}{c}d{x}	\int_{a}^{b}{c}d{x}
10	$\sum_{a=b}^{c} x$	Summation	4	sum{a=b}{c}{x}	\sum_{a=b}^{c}{x}
11	$\prod_{a=b}^{c} x$	Product	4	prod{a=b}{c}{x}	\prod_{a=b}^{c}{x}
12	$\sqrt{a}$	Square Root	1	sqrt{a}	\sqrt{a}
13	$\sqrt[a]{b}$	Root	2	sqrt[a]{b}	\sqrt[a]{b}
14	|a|	Absolute Value	1	|a|	|a|
15	$\log a$	Logarithmic	1	log{a}	log{a}
16	$\log_{a} b$	Logarithmic with base	2	log{a}{b}	log_{a}{b}
17	$\ln a$	Natural Logarithmic	1	ln{a}	ln{a}
18	$\lim_{a \to b} c$	Limit	3	lim{a rightarrow b}{c}	\lim_{a\rightarrow b}{c}
19	a mod b	Modulus	2	a mod b	a \bmod b

20	sin(a)	Sine	1	sin{a}	sin{a}
21	cos(a)	Cosine	1	cos{a}	cos{a}
22	tan(a)	Tangent	1	tan{a}	tan{a}
23	a'	Derivative	1	{a}'	{a}'

Table 5 - The distinct mathematical operations. There is alternative notation to describe the product operation-in our implementation we only make use of the one with the asterisk sign.

4.3 The Formal Grammar

In this section we define the designed formal grammar we propose for this work; the set of the production rules include all the operations which were presented in the previous chapter.

In the very beginning, we need to make clear what a mathematical expression in a technical document is. It is a finite combination of symbols that is well-formed according to rules that depend on the context. Mathematical symbols can designate numbers (constants), variables, operations, functions, brackets, punctuation, and grouping to help determine order of operations, and other aspects of logical syntax. A mathematical expression is a syntactic construct which should be well-formed, and the operators must have the correct number of inputs in the correct places. Therefore, strings of symbols that violate the rules of syntax are not well-formed and are not considered as valid mathematical expressions. For example, the expression $1 + 2 \times 3$ is well-formed, but the expression 9×4)x+/y is not.

Every mathematical formula is composed by a "kernel". This kernel represents all the numbers, variables, operators, and delimiters. Our definition of the formal language models the generation of mathematical formulas using the different mathematical elements constructing it.

Numbers

Numbers may be positive or negative, where when the sign is missing, the positivity of the number is assumed. Examples of numbers found in mathematical formulas are: 1, 2, 3.5, 10.999, -0.81, 0, π, e. The two latter examples are universal constants and are used in places of numbers. We also need to mention the infinity sign which may be found in many mathematical expressions. This does not belong to the numbers set, but it describes a quantity and is widely used.

Variables

Variables are Latin alphabet (and sometimes Greek alphabet-we only consider Latin alphabet letters at this point of time) letters which stand for numerical values in a mathematical expression. Occasionally, variables have names that are formed by more than one letters, constituting a word representing a variable. Some examples of variables are: x, y, z, A, B, ratio, median.

Operators

All the rest of the symbols that signify relationships and operations among numbers or variables are called operators. When describing a mathematical expression in the LaTeX format, a number of keywords are also used to indicate different mathematical functions (e.g. 'frac' for fraction, 'int' for integral). Other operators in a mathematical formula may be +, -, /, *, =, >, <, !.

Delimiters

Delimiters are the punctuation marks used in mathematics and are used to signify where a mathematical expression ends and another one begins. The most widely used one is parenthesis, but brackets ({ , }) and square brackets ([,]) are also used infrequently.

When reading a mathematical formula, the elements forming it may be compared to the words assembling a natural language sentence as the mathematical operators may take the place of the verbs and operands are the substitutes of the nouns. Formulas and equations follow the standard grammatical rules that apply to words; therefore, mathematical symbols can correspond to different parts of speech. For instance, $1+2=3$ is a perfectly good complete sentence.

The symbol "=" acts like a verb. Below are a couple more examples of complete sentences. Further examples may be the expressions $3xy < -2$ and $5z \in R$. On the other hand, an expression like $2x-10y$ is not a complete sentence as there is no verb. Such expressions should be treated as nouns.

The proposed formal language also provides a method for the synthesis of different mathematical functions. We define the Grammar of the FL as G=(N, T, S, P), where:

- N is the non-empty, finite set of the non-terminal symbols. Non-terminal symbols are illustrated with capital letters and can only be found on the left side
- T is the finite set of the terminal symbols. The symbols that are not in the non-terminal set, are called terminal symbols or alphabet

symbols and they are the symbols that make up the strings in the language. No rule can be applied to these symbols.

- S stands for the start symbol of the Grammar. This is the special symbol required for each application of rules to begin the derivation of strings in the language. Subsequently, the only grammatically correct strings for a given grammar are the strings that can be derived by rule applications from the start symbol.

- P is the corpus of the production rules, presented in Section 4.4.

Finite languages are those containing only a finite number of words and they are regular languages, as one can create a regular expression that is the union of every word in the language. Every finite set represents a regular language. The purpose of a regular grammar is to specify how to form grammatically correct strings in the language the grammar represents. In our system, a regular language is applied for the generation of syntactically correct mathematical formulas.

The execution of the production rules is a recursive procedure where each non-terminal symbol is assigned a concatenation of a number of terminal symbols [36]. The execution of a number of the production rules in order to form and output a sentence , is called production. By using the formal language, we focus on the syntactical aspect of the language, meaning the internal structural patterns of it. Through every production, a different selection of production rules is made, based on randomness, which results in a totally different mathematical formula. At this point of the generation process, it is obvious that the formal grammar generates sentences that do not make semantic sense, but they are syntactically correct, following the syntax of mathematics. As a result, the grammar that was developed is able to generate any sentence, which may or may not have a semantical meaning. Due to this vagueness of the generated objects, the semantically valid mathematical formulas are assessed during the second step of the processing. During that layer of processing, a number of regulations are stipulated, to distinguish the formulas that have a meaning, from the ones that are not valid and would never appear in a scientific document.

4.4 Production rules

The set of the production rules describe the way that the words will be arranged in a sentence and each rule describes the way that the symbols may be replaced. In our grammar, we aim to generate formulas which may be found in scientific documents, so, the format of the generated formula will be of type

<expression><symbol of relation><expression>. In fact, this is the format to which our start symbol leads.

1. S -> EXP SIGN EXP

2. EXP -> FACTOR | EXP BINOP FACTOR | NEGOP EXP

3. BINOP -> PLUS | MINUS | TIMES

4. NEGOP -> MINUS

5. SIGN -> EQS | GRS | LSS | GOES | LOES

6. FACTOR -> VAR | NUMBER | PARENTHESIS | FRACTION | SQROOT | ABS | EXPONENTIAL | FACTORIAL | LIMIT | INTDEF | INTINDEF | LOGARITHM | LOGNAT | SUMMATION | PRODUCTION | ROOT | SINE | COSINE | TANGENT | MODULUS | DERIVATIVE

7. VAR -> LETTER | LETTER VAR | BCSL PI | BCSL EPSILON

8. LETTER -> 'a' | 'b'| 'c' |'d' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

9. PARENTHESIS -> LP EXP RP

10. FRACTION -> BCSL FRAC LBR EXP RBR LBR EXP RBR

11. ABS -> VB EXP VB

12. ROOT -> BCSL SQRT LSQBR INTEGER RSQBR LBR EXP RBR

13. SQROOT -> BCSL SQRT LBR EXP RBR

14. EXPONENTIAL -> EXP CARET LBR EXP RBR

15. FACTORIAL -> SEPEL EXM

16. INTDEF -> BCSL INT US LBR EXP RBR CARET LBR EXP RBR LBR EXP RBR DIFF LBR EXP RBR

17. INTINDEF -> BCSL INT LBR EXP RBR DIFF LBR EXP RBR

18. LIMIT -> BCSL LIM US LBR EXP BCSL RARROW EXP RBR LBR EXP RBR

19. SUMMATION -> BCSL SUM US LBR VAR EQS EXP RBR CARET LBR EXP RBR LBR EXP RBR

20. PRODUCTION -> BCSL PROD US LBR EXP EQS EXP RBR CARET LBR EXP RBR LBR EXP RBR

21. LOGARITHM -> BCSL LOG US LBR EXP RBR LBR EXP RBR | LOG LBR EXP RBR

22. LOGNAT -> BCSL LN LBR EXP RBR

23. SINE -> BCSL SIN LBR EXP RBR

24. COSINE -> BCSL COS LBR EXP RBR

25. TANGENT -> BCSL TAN LBR EXP RBR

26. MODULUS -> SEPEL BCSL MOD SEPEL

27. DERIVATIVE -> SEPEL DERS

28. NUMBER -> INTEGER | INTEGER DOT INTEGER | BCSL INFS

29. INTEGER -> NUMERIC | INTEGER NUMERIC

30. NUMERIC -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

31. PLUS -> '+'

32. MINUS -> '-'

33. TIMES -> '*'

34. MOD -> 'bmod'

35. LP -> '('

36. RP -> ')'

37. LBR -> '{'

38. RBR -> '}'

39. LSQBR -> '['

40. RSQBR -> ']'

41. EQS -> '='

42. GRS -> '>'

43. LSS -> '<'

44. GOES -> '>='

45. LOES -> '<='

46. VB -> '|'

47. EXM -> '!'

48. FRAC -> 'frac'

49. LOG -> 'log'

50. LN -> 'ln'

51. INT -> 'int'

52. CARET-> '^'

53. SQRT -> 'sqrt'

54. LIM -> 'lim'

55. SIN -> 'sin'

56. COS -> 'cos'

57. TAN -> 'tan'

58. SUM -> 'sum'

59. PROD -> 'prod'

60. DERS -> '''

61. DOT -> '.'

62. RARROW -> 'rightarrow'

63. US -> '_'

64. DIFF -> 'd'

65. INFS -> 'infty'

66. BCSL -> '\'

67. PI -> 'pi'

68 EPSILON -> 'epsilon'

69. SEPEL -> VAR | PARENTHESIS

Detailed description of production rules introduced above:

1) The very first rule to be defined is the production of the start symbol 'S', which signifies the point at which the generation of the string will begin. At this point, it is made sure that the format of the formula will be a mathematical expression related to a second mathematical expression through a symbol that defines their relationship. The symbol may describe an equality or inequality between the two expressions, it is indicated as SIGN in this grammar and is thoroughly described by rule 5.

2) The second rule describes the definition of mathematical expressions: a mathematical expression consists of mathematical numbers or variables or a combination of them, using operations such as addition, subtraction,

multiplication etc. For a better legibility of the terms in the formal grammar definition, we define as a `factor` any mathematical element that can have a value prior or post calculation. This definition is better described in rule number 5.

This present rule has a second part on the right side, which also includes the left part of the rule, which makes it a recursive rule, when existing mathematical expressions are combined with factors through operations, as mentioned above. The operators connecting these factors are binary operators that require two operands for the evaluation. In order to include the negative numbers, rule 4 was also introduced and is described below. For the formation of a negative number, the number follows the minus symbol. This rule also takes under consideration expressions that have a negative sign in the front, as the symbol EXP can indicate a factor of an expression, which may be a variable, a number or any mathematical operation.

3) The third rule includes the binary operators that can be found in any mathematical expression. PLUS, MINUS, and TIMES, stand for the signs of the addition, subtraction, and multiplication, respectively. The operation of division is left outside of this category due to the different syntax in the $\LaTeX$ format, which assorts it in fractions.

4) NEGOP stands for the negation operator and is defined in rule 4 for the formation of negative values. This the minus sign that is generated by rule 32 further to this rule.

5) This rule contains all the relation signs that can be occurred between any two expressions, according to rule 1. More specifically, this rule introduces that the sign connecting two expressions can be among the “equal”, “greater than”, “less than”, “greater than or equal to” and “less than or equal to” signs, represented by the symbols EQS, GRS, LSS, GOES and LOES, respectively.

6) FACTOR symbol describes all the different math elements which may occur in a mathematical expression, including numbers and variable names. The elements participating in the right part of this production rule represent the mathematical operations that were listed in Table 5, excluding the first four operations-which are the binary operations defined in rules number 2 and 3.

    - VAR stands for a variable name, which can be a factor of an expression or stand as a full expression by itself. The definition of VAR symbol is found in rule 7.

- NUMBER represents any umber that is eligible for participating in a mathematical expression or forming one. The definition of NUMBER symbol is described in rule 28.
- Parentheses in a mathematical expression are handled as a distinct entity of an expression which includes some expression in it, despite the fact that they are not listed as an operation. The description of symbol PARENTHESIS is defined further in rule 9.
- The rest of the symbols in this production are the mathematical operations included in the Table 5, with the correspondence of one symbol per operation.

7) Every variable in a mathematical expression can be named by a letter or a word-which is a concatenation of letters. Based on this knowledge, rule 7 is defined: each variable, represented by the grammar symbol VAR, is a letter or letters following one each other in a string. Thus, the first part of the production is used to represent the one-letter variables, and the second part to include variables which their names are words. Again, this production uses recursion to generate and illustrate words formed by letters. One of the elements of this present production is number pi, which is a mathematical constant and is represented using Greek letter Pi, or epsilon.

8) The letters used in rule 7 for the formation of word variables, but also to describe variables named after a letter, are provided in this production rule. The symbol used to describe every each one of these letters is LETTER.

9) As mentioned in rule 6, parentheses are treated as distinct operations in a mathematical expression, by convention. This is a way to clarify that inside each pair of parenthesis signs, an expression is found, and so parenthesis sign pair is expected to be empty. This production rule, when executed, makes use of rules 35 and 36, where the terminal signs of left and right parentheses are defined. Additionally, the second argument of this rule derives from rule 2, where an expression is defined. Therefore, it is clear that an instance of a parenthesis is formed by the left parenthesis sign, an expression as the component of the parenthesis, and a right parenthesis sign, in this order.

10) Because of the LaTeX format we are using for the format of the formulas generated, all the operations described in an expression must follow this format. For a fraction representation, the symbol string representation in LaTeX is \frac{input1}{input2}, where input1 is the numerator and input2 is the denominator of the fraction. Subsequently, this production rule is formed by the concatenation of a backslash (rule 66), symbol of fraction,

FRAC (rule 48), left bracket (rule 37), the symbol string of the expression representing the numerator of the fraction (rule 2), right bracket (rule 38), left bracket (rule 37), the symbol string of the expression representing the denominator of the fraction (rule 2), and a right bracket (rule 38). Any expression (either number, variable, or mathematical operation) may consist the numerator or the denominator of the fraction, this is why the symbol of expression (EXP) is used.

11) We use ABS symbol to describe the absolute value of an expression. Again, the symbol of expression is used to indicate that any expression can be the interior of an absolute value operation. So, production 11 defines the absolute value of an expression as the concatenation of two vertical bars with a symbol string of any expression in between (rule 2). Vertical bar is a terminal symbol defined in rule 46.

12) Again, based on the LaTeX format, a root is described as \sqrt[degree]{input} so this is the format we will use as well. Based on this, in rule 12, a square root operation is defined as the concatenation of a backslash (rule 66), symbol of square root, SQRT (rule 53), left square bracket (rule 39), an integer symbol (rule 29) as the degree of the root, right square bracket (rule 40), left bracket (rule 37), the symbol string of the

expression representing the inner expression of the root as an expression (rule 2), and a right bracket (rule 38).

13) In many cases, roots are illustrated missing the degree number. These are the cases where a square root is presented, and we handle them as a different operation. The LaTeX format of a square root is \sqrt{input}. Based on this, in rule 13, a square root operation is defined as the concatenation of a backslash (rule 66), symbol of square root, SQRT (rule 53), left bracket (rule 37), the symbol string of the expression representing the inner expression of the square root, and a right bracket (rule 38).

14) Powers of values and expressions are described as exponential operations in rule 14. The expression that forms the base of the exponential may be a variable name or a number, or even any expression. In the case of an expression as a base of the exponential, the expression is expected to be within a parenthesis, as the math notation requires. To make sure that our grammar complies with this rule, production 69 is used to define the separate elements that may form the base of an exponential operation. Whereupon, an exponential component is defined as the concatenation of the base (rule 69), the caret sign (rule 53), left bracket (rule 37), the symbol

string of the expression representing the exponent expression (rule 2), and a right bracket (rule 38).

15) Factorials are also required to be applied on an expression of a single number or variable, or an expression which will be demarcated by parentheses, so instead of using the EXP symbol, we will use the SEPEL (rule 69) as well. The syntax of a symbol string describing an exponential operation is a SEPEL symbol (variable or parenthesis) followed by an exclamation mark symbol (rule 47).

16) For the definite integrals, we introduce production rule 15, where the arguments required to fully define the integral are given, again, in a specific order, after the LaTeX format. Therefore, the symbol INTFIN, representing a definite integral operation, consists of the concatenation of the backslash (rule 66), the symbol of the integral (rule 51),an underscore (rule 63) to signify the lower bound of the integral, a left bracket (rule 37), the symbol string of the expression representing the lower bound (rule 2), a right bracket (rule 38), the caret sign (rule 53) to indicate the upper bound of the integral, a left bracket (rule 37), the symbol string of the expression representing the upper bound (rule 2), a right bracket (rule 38), a left bracket (rule 37), the symbol string of the expression inner the integral

operation (rule 2), a right bracket (rule 38) to indicate the end of the inner-main expression, the letter ‘d’ (as a terminal symbol introduced in rule 64) used along the differential factor of the integral, and finally, the differentiator as an expression (rule 2), again within brackets (rules 37 and 38). The above syntax refers to the symbol string \int_{lower bound} ^ {upper bound}{inner expression} d {differentiator} in LaTeX.

17) Regarding indefinite integrals, the production follows the same pattern, with the lower and upper bounds of the integral missing. Therefore, as the syntax of an indefinite integral is \int{inner expression}d{differentiator} in LaTeX format, this production rule is formed by the concatenation of a backslash (rule 66), the symbol of the integral (rule 51), a left bracket (rule 37), the symbol string of the inner expression of the integral (rule 2), a right bracket (rule 38) to indicate the end of the inner-main expression, the differential symbol, a left bracket (rule 37), the expression inner the integral operation, a right bracket (rule 38).

18) Rule 18 defines the limits, which in LaTeX have the syntax of \lim_{expression1 \rightarrow expression2}{main expression}, that describes the limit of the expression “main expression” when expression1

"approaches" expression2. Although we could assume that expression1 is a variable, and expression2 is either a variable or a number(including infinity symbol), as most limit operations tend to be defined, in the context of the formality of the grammar, we assign these two expressions as any expression, not limiting their content to a variables and number. Later, though, we are going to eliminate elements with high complexity in these argument positions.

19) Summation operation is formed as the concatenation of the backslash (rule 66), the symbol of the summation, SUM (rule 58),an underscore (rule 63) to signify the starting point of the summation, a left bracket (rule 37), the variable symbol based on which, the summation will operate (rule 7), the equality sign symbol (rule 41), an expression which value will be assigned initially to the variable (rule 2), a right bracket (rule 38), the caret sign to indicate the ending point of the operation (rule 53), a left bracket (rule 37), the symbol string of the expression representing the ending point of the operation, a right bracket (rule 38), a left bracket (rule 37), the symbol string of the expression inside the summation, and a right bracket (rule 38) to indicate the end of the main expression. Again, at this point, and because of the definition of the context-free grammar we are introducing, any

component of a mathematical operation is assigned as mathematical expression, which makes a set of unlimited terms prospective for any argument of an operation.

20) On the same base as the summation operation, production operation requires exactly the same arguments as an input to be completely defined. Consequently, as the LaTeX format of a production operation is \prod_{variable= expression1}^{expression2}{main expression}, the execution of rule 20 that defines this present operation will output the concatenation of the following symbols: the backslash (rule 66), the symbol of the production, PROD (rule 59),an underscore (rule 63) to signify the starting point of the production, a left bracket (rule 37), the variable symbol based on which, the production will operate (rule 7), the equality sign symbol (rule 41), an expression which value will be assigned initially to the variable (rule 2), a right bracket (rule 38), the caret sign to indicate the ending point of the operation (rule 53), a left bracket (rule 37), the symbol string of the expression representing the ending point of the operation, a right bracket (rule 38), a left bracket (rule 37), the symbol string of the expression inside the production, and a right bracket (rule 38) to indicate the end of the main expression.

21) This rule describes the format of the logarithm symbol in our language. For the complete definition of a logarithm, two arguments are required, the base of the logarithm and the mathematical expression within the logarithm. For the formation of the LATEX format of the logarithmic operation, we use the sequence of a backslash (rule 66), the symbol of the logarithm, LOG (rule 49), an underscore (rule 63) to signify the base of the logarithm, a left bracket (rule 37), the symbol string of the expression representing the base (rule 2), a right bracket (rule 38), a left bracket (rule 37), the symbol string of the expression inside the logarithm (rule 2), and a right bracket (rule 38). Alternatively, the logarithm operation may occur without an argument defining the base. In this case, the syntax is as follows: a backslash (rule 66), the symbol of the logarithm, LOG (rule 49), a left bracket (rule 37), the symbol string of the expression inside the logarithm (rule 2), and a right bracket (rule 38). The occurrences where the base is not provided refer to cases in which no confusion is possible, because of the context given, or in cases where the argument of the base does not matter.

22) For the natural logarithm, the LATEX format for its representation is \ln{expression}, so we define it as the concatenation of a backslash (rule 66), the symbol of the natural logarithm, LN (rule 50), a left bracket (rule

37), the symbol string of the expression inside the natural logarithm (rule 2), and a right bracket (rule 38).

23) Rule 23 explains the production of the trigonometric operation of sine. In LaTeX format, sine is represented as a backslash followed by the 'sin', which is followed by the operation, which sine is to be evaluated, in brackets. Therefore, our symbol representation is the concatenation of the symbols of a backslash (rule 66), the symbol of the sine operation, SIN (rule 55), a left bracket (rule 37), the symbol string of the expression inside the sine operation (rule 2), and a right bracket (rule 38).

24) Identically to sine operation, cosine has the same syntax, which makes rule 24 to generate a word consisting of a backslash (rule 66), the symbol of the cosine operation, COS (rule 56), a left bracket (rule 37), the symbol string of the expression inside the cosine operation (rule 2), and a right bracket (rule 38).

25) Similarly to the previously introduced trigonometric operations, tangent is described in rule 25, where its LaTeX format is \tan{expression}. This production rule, therefore, defines the word of the tangent as the sequence of a backslash (rule 66), the symbol of the tangent operation, TAN (rule

57), a left bracket (rule 37), the symbol string of the expression inside the tangent operation (rule 2), and a right bracket (rule 38).

26) Binary operator modulus in our grammar is introduced as a distinct mathematical operator, rather than including it in the rule of binary operators (rule 3). Two expressions (in this case, we assume that the two arguments of the modulus operation may be variables or expressions within parentheses, so that the expression's component is delimited by the parenthesis signs) are connected, with the modulus symbol (defined in rule 34) between them. This concatenation forms a symbol string describing the modulus operation between two expressions.

27) First order derivatives of expressions are described by the expression followed by an apostrophe, which is the exact same syntax used in LaTeX. In rule 27, the syntax of a first order derivative of an expression (symbol EXP) is described as the sequence of a separate element symbol (SEPEL), describing a variable or an expression in a parenthesis, followed by an apostrophe.

28) To fully define a number as a quantity in our formal language, we define rule 28 which defines mathematical quantities in a more abstract way, as it

includes numeric values but also the infinity sign. Therefore, the NUMBER symbol incorporates integer and real numbers (composed of integer numbers split by a dot, to define the integer from the decimal part of the number), and the infinity symbol.

- INTEGER : this symbol is described in rule 29 and defines any integer type of a value participating in the formula.
- INTEGER DOT INTEGER : this sequence of symbols describes a real number, where the integer part constitutes of an integer symbol (defined in rule 29), followed by a decimal separator (the DOT symbol defined in rule 61 standing for the decimal point), and, again, an integer symbol that will now stand for the fractional part of the number.
- BCSL INFS : for the representation of the infinity symbol, the LATE X format requires a backslash followed by 'infty' which is a keyword for the infinity symbol. The terminal symbol 'infty' is given by rule 65 for the symbol INFS (standing for infinity symbol).

29) This rule is a recursive production of an integer number, where an integer may be any one-digit number represented by a numeric symbol (0-9), as

designated in rule 30, or a concatenation of numeric symbols with an integer, for the representation of integers of two or more digits.

The terminal symbols introduced in rules 30-67 are the letters forming the words of the language we introduce. Consequently, every word that may occur in a sentence of this formal language must be a concatenation of the terminal symbols described above.

4.5 Example

After every execution of the formal grammar using the production rules, the output is a new mathematical formula in a LaTeX format. For a better understanding of the previously introduced grammar rules, we present an example of a randomly generated mathematical formula symbol string along with its optical representation and the tree illustrating the production rules executed for the generation of the formula. The term randomly in this case indicates that any random combination of the introduced production rules will result to a well-formed mathematical expression. The randomly generated mathematical formula, making use of the formal grammar introduced above, is:

```
j > \frac{ 3 ^{ z } }{ | c | }
```

and when this symbol string is compiled through LATEX, the output is the optical representation of the formula, which is:

$$j > \frac{3^z}{|c|}$$

This output is a result of a sequential execution of production rules of our grammar, which is initiated by the execution of the production of the start symbol S, and for every non-terminal symbol of the grammar, the execution moves forward until a terminal symbol is reached and is mounted on the sentence as a word. The image below shows the execution tree whose leaves are the words of the generated sentence in our language. The final form of the sentence is a concatenation of all the leaves of the tree (which are the terminal symbols), from left to right.

At this point of our work, it is vital to mention that the manual execution of the production rules can output to any possible mathematical formula, though, when executed in an application, the loop of the execution can go under a big number of recursions, which is a problem that can cause issues considering the memory usage.

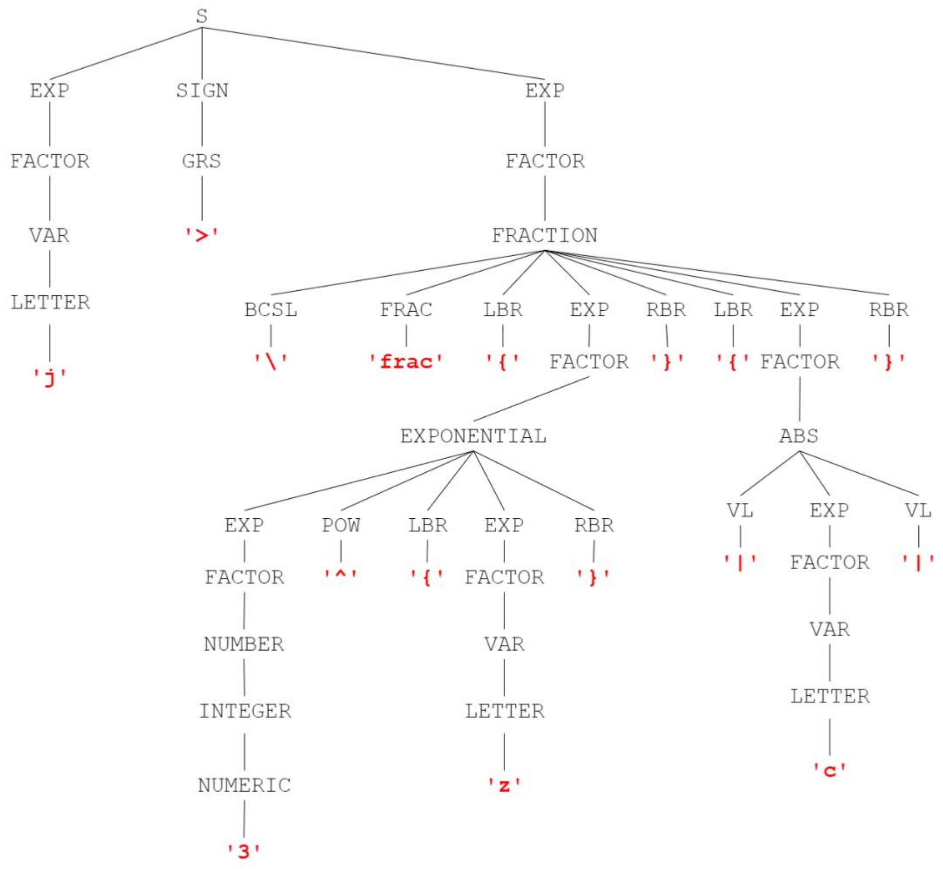


Figure 13 - The tree representation of the execution of the production rules that output the symbol string j > \frac{ 3 ^{ z } }{ | c | }. The execution terminates when all the symbols in the symbol string are lowercase, indicating that they are all terminal symbols. Similarly, to the presented example, more complex MF can be formed.

4.6 Syntactical and Logical Restrictions

Each generated mathematical formula is based on the execution of the production rules making our grammar. The grammar is a tool that ensures that the outcome of the rule execution will be a syntactically valid sentence, which in our case is going to be a mathematical formula, taking no notice of the semantical correctness of it. Consequently, it is not necessary that the newly formed formula will combine mathematical elements that their associations will make sense. The generated formula is forwarded to a second layer of processing. During this second step, every formula which was generated using the formal grammar undergoes a set of checks. There are several rules to define the validity of the generated formulas in terms of semantics and syntactical optimization. The checks that the formula goes under are in a form of constrains that describe any mathematical formula found in a scientific document in the field of engineering. By this, it is clarified that formulas describing theoretical mathematics are out of our area of interest.

The checks are critical because they aim to preserve the integrity of the formula regarding the syntax of it, and they are presented as restrictions on the formulas:

- Syntactical restrictions

The restrictions that try to eliminate the redundant notation which was caused by the grammar rules and does not supply any additional functionality to the elements of the formula. Table 6 illustrates the different cases where excess notation was detected in the formula, and representative examples are provided.

- Logical restrictions

These restrictions deal with undefined forms of mathematical expressions and indeterminate forms and values in the formulas that when found in specific positions in the formula, the formula does not make sense. Logical restrictions are presented in Table 7, along with examples that embody each distinct case.

Syntactical Restrictions			
Restr. No	Description	example 1	example 2
S_1	Redundant parentheses	$y > \left(\frac{x}{3!}\right)$	$\|(x-1)\|$
S_2	Double parentheses	$((x-y))$	
S_3	Double absolute value signs	$\|\|x-y\|\|$	

Table 6 - The syntactical restrictions in a generated formula.

Logical Restrictions			
Restr. No	Description	Example 1	Example 2
L_1	Low border greater than high border	$\int_{4}^{1} \frac{x^3-2}{x!} dx$	
L_2	Variable in differential not in function of integral	$\int_{4}^{1} \frac{x^3-2}{x!} dy$	

L_3	Variable in start value not in argument of summation	$\sum_{n=1}^{10} \frac{\alpha_k}{2}$	
L_4	Starting point in summation greater than upper limit	$\sum_{n=10}^{2} \alpha_k$	
L_5	Variable of limit "approaching" a value not in the function of the limit	$\lim_{x \to 0} \frac{y-1}{y+1}$	
L_6	False inequality	$3 < 1$	$4 = 9$
L_7	Square root principal: value inside the root must be non-negative	$\sqrt{-2}$	
L_8	Logarithm principal: value inside the logarithm must be positive	$\log(-7)$	$\log(0)$
L_9	Logarithm principal: value inside the natural logarithm must be positive	$\ln(-2)$	$\ln(0)$
L_{10}	Indeterminate forms	$\infty + (-\infty)$	$0*(\infty)$
L_{11}	Undefined forms	$\frac{0}{0}$	$\frac{x}{0}$

L_{12}	Indeterminate forms	$\frac{\infty}{0}$	$\frac{\infty}{\infty}$
L_{13}	Indeterminate forms	1^{∞}	∞^{0}

Table 7 - The logical restrictions in a generated formula.

In the case that a syntactical restriction is detected in the generated formula, term which violates the restriction may be modified, and the generated formula may process to the next step of the procedure. When a logical restriction is occurred, the formula is considered invalid, and is rejected – will not be taken under consideration for further processing.

Let us consider an example of a generated mathematical formula such as

$$y = \int_0^k \frac{1}{|k-2|} dy. \qquad (2.1)$$

We can see that this formula is syntactically correct. The variable of the differential of the integral (element y), though, does not occur in the inner function of the integral, which does not make sense for an integral declaration. This is also what rule L_2 indicates, making the formula invalid. As a result, this generated formula is going to be rejected.

For a second example, we have the formula 2.2 below, which also includes an integral. In this example, there are many restrictions violated.

$$a = \int_{99}^{3} \frac{1}{|(k-2)|} dx \qquad (2.2)$$

According to S_1, the denominator of the function inside the integral has redundant parenthesis, which, if missing, the context of the formula would not be affected. Logical restrictions are eligible to transform the formula to a more simplified one, rejecting redundant elements inside the formula. After processing the formula 2.2, applying the rule S_1, the unneeded set of parentheses will be removed, so the formula will turn into 2.3.

$$a = \int_{99}^{3} \frac{1}{|k-2|} dx \qquad (2.3)$$

There are no further syntactical restraints to be violated in this example, so we will now check the logical restrictions. Regarding the logical restrictions, there are L_1 and L_2 to be infringed. When logical rules are violated, the generated formula may not be transformed to a valid one, and it simply gets rejected. Therefore, for the execution of the syntactical and logical rules, we will initially check for the logical restraints in a formula. In the case that there is no violation of them, the execution may proceed to the check of the syntactical restraints, otherwise, the formula will be rejected directly.

4.7 Generated Formulas

In this section, we present examples of formulas that were generated by the execution of the production rules. Each generated sentence of the formal language is formed by symbols that form the encoding of a mathematical expression in LaTeX, and the optical representation of the formulas is also provided. The examples are collected in Table 8.

Generated formula visualized in LaTeX	Valid
$\lvert\frac{\frac{\lvert r*c+\frac{e}{9}\rvert}{u}}{\lvert 6\rvert}\rvert > b$	Yes
$h >= g - 3$	Yes
$c < \int_6^v \frac{6^2}{\int_2^w \frac{x^c}{\sqrt{s}} da} dh$	No
$b >= x^{\lvert\lvert o\rvert\rvert}$	Yes
$v <= \sum_{d=4}^{7} \sin k$	No
$d > \frac{k^x}{\lim_{6\to l} \sum_{x=5}^{9} \sqrt{m}}$	No

$z = \sum_{r=7}^{2} \int_{1}^{b} \frac{2^{\infty}}{\sqrt{s}} dt$	No
$j > \sqrt{\lim_{l \to 3} \sum_{q=4}^{0} \frac{\sin z}{\sin 0}}$	No

Table 8 – In this table we present examples of randomly generated mathematical formulas and evaluate their logical validity according to the rules presented above. Any generated formula will be rejected as invalid if one or more occurrence of constraints from Table 7 is applied to the formula.

5. Detecting and Recognizing of Pseudo-Algorithms in Scientific Documents and SPN representation

5.1 Introduction

Technical documents are formed by several modalities including plain text, diagrams, tables, algorithms, images etc. Algorithms have a significant place in technical documents not limited in publications in the field of computer science and software engineering, but moving further to computer vision, bioinformatics etc. In this work we present the extraction and recognition of algorithms and their components in scientific technical documents. The intend here is the processing of any image illustrating a page of a technical document to detect any algorithmic component inside the image and analyze it for understanding purposes. In other words we aim to detect algorithmic components in an image of a technical document based on their structural features in the text and represent them in a Stochastic Petri-net form for evaluating its functional behavior. The process is divided into two different processing parts. The first one is the detection and the extraction of the algorithmic components in a document and the second one is the translation of it into a graph and its SPN representation. Specifically, the first part consists of two steps; the first one is the detection of the sections of the document that describe the algorithm. In the second step we perform image processing so we can extract information about the component of it and proceed to the recognition of the algorithm. Finally, in the second part, the generation of the graph for the algorithm and its SPN mapping is described in an effort to express the algorithm first level functional associations.

5.2 Optical Detection of Pseudo-codes in Documents

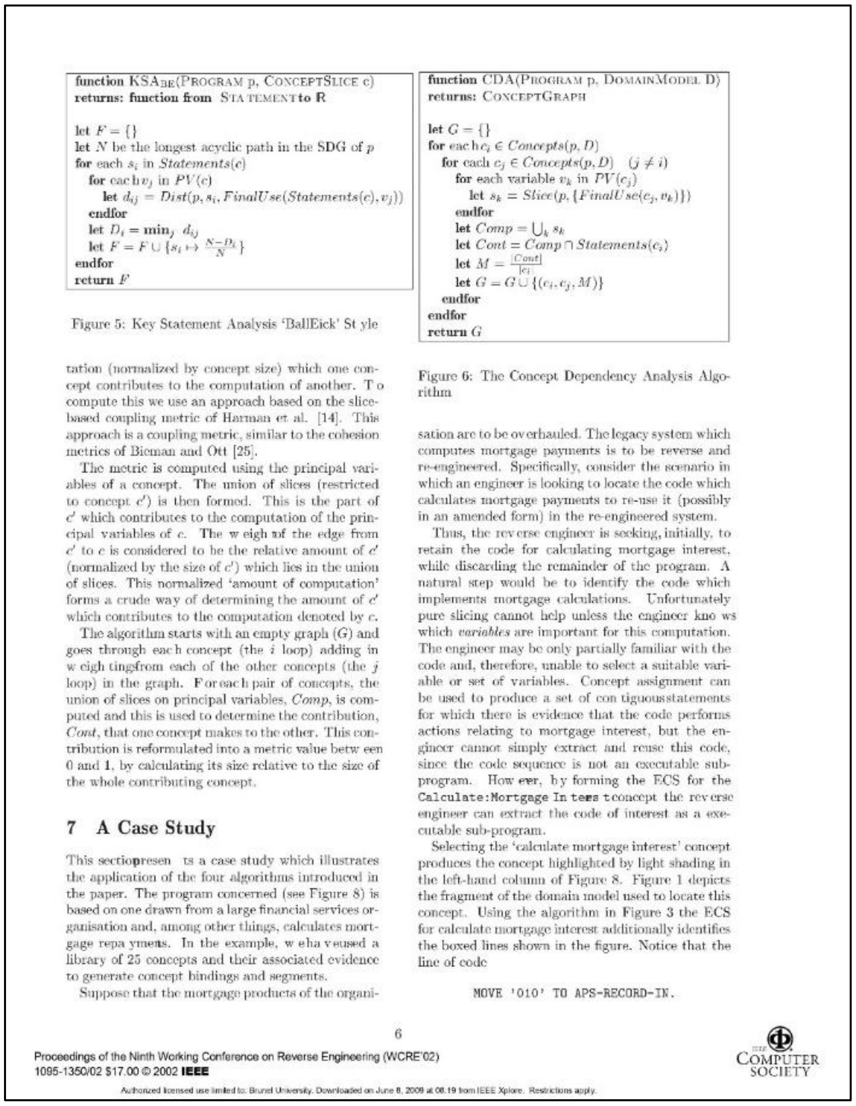
5.2.1 Extraction of different text blocks in documents

It is always an interesting idea to follow a recognition process that attempts to simulate the way the human brain interprets the algorithmic components in the documents. Thus, here we attempt to follow such an approach that attempts to emulate the detection and recognition of pseudo algorithms in technical documents at a high level of representation. The first step towards this simulation is the extraction of the distinct components in the document, so that we can then recognize whether each component describes an algorithm in it, or not. This process is part of the pre-processing of the input document, as the images representing the entire technical document will undergo different layers of modifications. These images are further segmented into blocks to be examined for algorithmic components. Thus, a pyramidal reduction scheme methodology can be used for the recognition and extraction of the various components of the document. According to this methodology, the image is subject to repeated smoothing and subsampling until we reach a point to which the individual structural parts of the image are distinguished [37]. For this purpose, a variety of different smoothing kernels may be used, mainly changing depending on the size of the font used in the text found in the document. Thus, here for the simulation of the pyramidal reduction, and in order to maintain all the information of the initial image, we simulate the result of a higher level of the pyramid using dilation. Dilation is one of the morphological transformations used on binary images where a kernel is used and based on its size, the area of the objects in the binary image increases. This way, the larger the dimensions of the kernel, the more the area that will be merged and included in a distinct block. After multiple tests on sample documents and based on the IEEE Standards which require normal text to be single-spaced in 10-point font, the kernel which would output the most accurate recognition results was a 7x6 sized kernel. This size of the kernel is large enough to ensure that the extracted blocks will not be one-word text blocks, but will contain sets of words, and at the same time it must be small enough not to perceive the whole document as a single block. Ideally, this size of kernel will return the input image split in blocks, where each block will represent the title, or a paragraph, a sequence of paragraphs, an image, or an algorithm etc.

5.2.2 Pyramidal image representation

A pyramid is a multi-scale image representation which is used for the detection of objects in images using different scales [38]. During this representation, the input image at its original size is located at the bottom of the pyramid, and in each next layer, the image of the previous layer is resized and smoothed using Gaussian blurring. Each image is progressively subsampled until it has reached a size (minimum size), where no further resizing is needed.

In the following images we can see the different layers produced through the pyramidal process. While moving towards the higher layers, it is notable that, although the details of the image are not available, there is additional information about the structure of the elements in the image, such as the number of text blocks existing in the input image in forms of headlines, paragraphs or pseudo-algorithm blocks [39].



```
function KSA$_{BE}$(PROGRAM p, CONCEPTSLICE c)
returns: function from STATEMENT to $\mathbb{R}$

let $F = \{\}$
let $N$ be the longest acyclic path in the SDG of $p$
for each $s_i$ in $Statements(c)$
    for each $v_j$ in $PV(c)$
        let $d_{ij} = Dist(p, s_i, FinalUse(Statements(c), v_j))$
    endfor
    let $D_i = \min_j\ d_{ij}$
    let $F = F \cup \{s_i \mapsto \frac{N - D_i}{N}\}$
endfor
return $F$
```

Figure 5: Key Statement Analysis 'BallEick' St yle

```
function CDA(PROGRAM p, DOMAINMODEL D)
returns: CONCEPTGRAPH

let $G = \{\}$
for each $c_i \in Concepts(p, D)$
    for each $c_j \in Concepts(p, D)$    $(j \neq i)$
        for each variable $v_k$ in $PV(c_j)$
            let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
        endfor
        let $Comp = \bigcup_k s_k$
        let $Cont = Comp \cap Statements(c_i)$
        let $M = \frac{|Cont|}{|c_i|}$
        let $G = G \cup \{(c_i, c_j, M)\}$
    endfor
endfor
return $G$
```

Figure 6: The Concept Dependency Analysis Algorithm

tation (normalized by concept size) which one concept contributes to the computation of another. T o compute this we use an approach based on the slice-based coupling metric of Harman et al. [14]. This approach is a coupling metric, similar to the cohesion metrics of Bieman and Ott [25].

The metric is computed using the principal variables of a concept. The union of slices (restricted to concept c') is then formed. This is the part of c' which contributes to the computation of the principal variables of c. The w eigh tof the edge from c' to c is considered to be the relative amount of c' (normalized by the size of c') which lies in the union of slices. This normalized 'amount of computation' forms a crude way of determining the amount of c' which contributes to the computation denoted by c.

The algorithm starts with an empty graph (G) and goes through eac h concept (the i loop) adding in w eigh tingfrom each of the other concepts (the j loop) in the graph. F or eac h pair of concepts, the union of slices on principal variables, *Comp*, is computed and this is used to determine the contribution, *Cont*, that one concept makes to the other. This contribution is reformulated into a metric value betw een 0 and 1, by calculating its size relative to the size of the whole contributing concept.

7 A Case Study

This sectiopresen ts a case study which illustrates the application of the four algorithms introduced in the paper. The program concerned (see Figure 8) is based on one drawn from a large financial services organisation and, among other things, calculates mortgage repa ymets. In the example, w eha veused a library of 25 concepts and their associated evidence to generate concept bindings and segments.

Suppose that the mortgage products of the organisation are to be ov erhauled. The legacy system which computes mortgage payments is to be reverse and re-engineered. Specifically, consider the scenario in which an engineer is looking to locate the code which calculates mortgage payments to re-use it (possibly in an amended form) in the re-engineered system.

Thus, the rev erse engineer is seeking, initially, to retain the code for calculating mortgage interest, while discarding the remainder of the program. A natural step would be to identify the code which implements mortgage calculations. Unfortunately pure slicing cannot help unless the engineer kno ws which *variables* are important for this computation. The engineer may be only partially familiar with the code and, therefore, unable to select a suitable variable or set of variables. Concept assignment can be used to produce a set of con tiguousstatements for which there is evidence that the code performs actions relating to mortgage interest, but the engineer cannot simply extract and reuse this code, since the code sequence is not an executable subprogram. How ever, b y forming the ECS for the `Calculate:Mortgage In tees t`concept the rev erse engineer can extract the code of interest as a executable sub-program.

Selecting the 'calculate mortgage interest' concept produces the concept highlighted by light shading in the left-hand column of Figure 8. Figure 1 depicts the fragment of the domain model used to locate this concept. Using the algorithm in Figure 3 the ECS for calculate mortgage interest additionally identifies the boxed lines shown in the figure. Notice that the line of code

```
MOVE '010' TO APS-RECORD-IN.
```

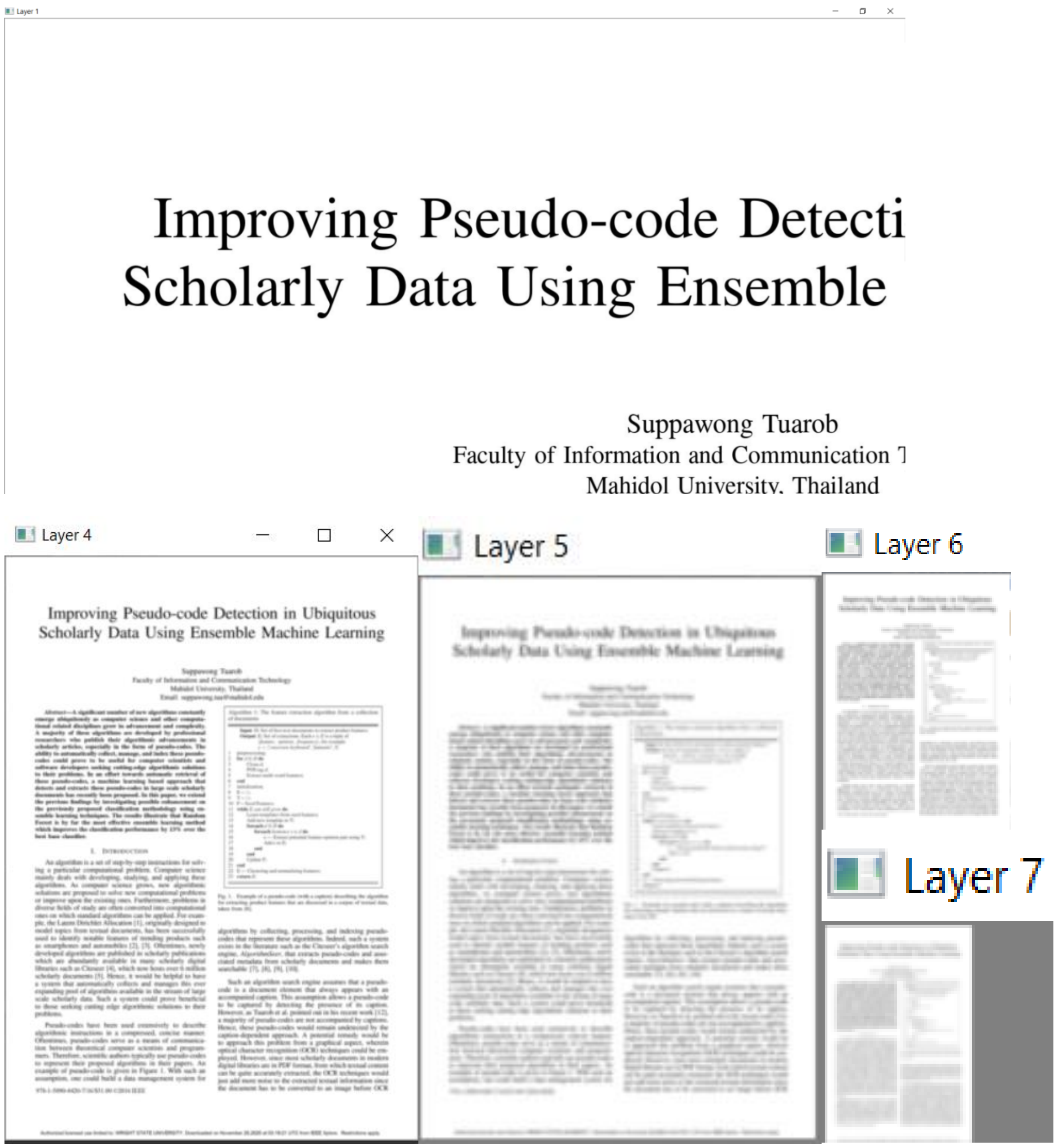
6

Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)
1095-1350/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Brunel University. Downloaded on June 8, 2009 at 08:19 from IEEE Xplore. Restrictions apply.

Image 1 - Example of input image as .jpg format (2550x3300)



Improving Pseudo-code Detecti
Scholarly Data Using Ensemble

Suppawong Tuarob
Faculty of Information and Communication T
Mahidol University, Thailand

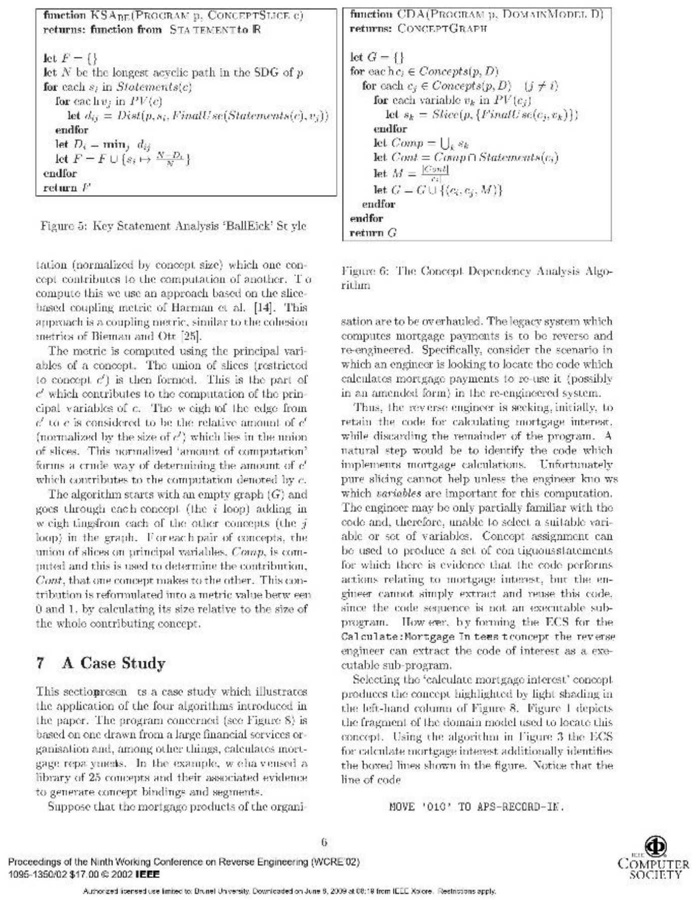
Improving Pseudo-code Detection in Ubiquitous
Scholarly Data Using Ensemble Machine Learning

Suppawong Tuarob
Faculty of Information and Communication Technology
Mahidol University, Thailand
Email: suppawong.tua@mahidol.edu

[illegible]

Image 2 - Different levels of text structure extraction after pyramidal transformation of input image

In order to achieve these results while still being capable of extracting information of the image component by using the morphological transformation of dilation. It is performed on greyscale images, and it preserves the shape of elements in the image, using a structuring element and a kernel for the transformation operation. Through dilation, the area of an element found in the input image is enlarged by gradually increasing the boundaries of the regions of the foreground pixels.



```
function KSA_BE(PROGRAM p, CONCEPTSLICE c)
returns: function from STATEMENT to ℝ

let $F = \{\}$
let $N$ be the longest acyclic path in the SDG of $p$
for each $s_i$ in $Statements(c)$
    for each $v_j$ in $PV(c)$
        let $d_{ij} = Dist(p, s_i, FinalUse(Statements(c), v_j))$
    endfor
    let $D_i = \min_j d_{ij}$
    let $F = F \cup \{s_i \mapsto \frac{N - D_i}{N}\}$
endfor
return $F$
```

Figure 5: Key Statement Analysis 'BallEick' St yle

```
function CDA(PROGRAM p, DOMAINMODEL D)
returns: CONCEPTGRAPH

let $G = \{\}$
for each $c_i \in Concepts(p, D)$
    for each $c_j \in Concepts(p, D)$   $(j \neq i)$
        for each variable $v_k$ in $PV(c_j)$
            let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
        endfor
        let $Comp = \bigcup_k s_k$
        let $Cont = Comp \cap Statements(c_i)$
        let $M = \frac{|Cont|}{|c_i|}$
        let $G = G \cup \{(c_i, c_j, M)\}$
    endfor
endfor
return $G$
```

Figure 6: The Concept Dependency Analysis Algorithm

tation (normalized by concept size) which one concept contributes to the computation of another. T o compute this we use an approach based on the slice-based coupling metric of Harman et al. [14]. This approach is a coupling metric, similar to the cohesion metrics of Bieman and Ott [25].

The metric is computed using the principal variables of a concept. The union of slices (restricted to concept c') is then formed. This is the part of c' which contributes to the computation of the principal variables of c. The w eigh tof the edge from c' to c is considered to be the relative amount of c' (normalized by the size of c') which lies in the union of slices. This normalized 'amount of computation' forms a crude way of determining the amount of c' which contributes to the computation denoted by c.

The algorithm starts with an empty graph (G) and goes through eac h concept (the i loop) adding in w eigh tingsfrom each of the other concepts (the j loop) in the graph. F or eac h pair of concepts, the union of slices on principal variables, $Comp$, is computed and this is used to determine the contribution, $Cont$, that one concept makes to the other. This contribution is reformulated into a metric value betw een 0 and 1, by calculating its size relative to the size of the whole contributing concept.

7 A Case Study

This sectionpresen ts a case study which illustrates the application of the four algorithms introduced in the paper. The program concerned (see Figure 8) is based on one drawn from a large financial services organisation and, among other things, calculates mortgage repa ymen ts. In the example, w eha veused a library of 25 concepts and their associated evidence to generate concept bindings and segments.

Suppose that the mortgage products of the organisation are to be ov erhauled. The legacy system which computes mortgage payments is to be reverse and re-engineered. Specifically, consider the scenario in which an engineer is looking to locate the code which calculates mortgage payments to re-use it (possibly in an amended form) in the re-engineered system.

Thus, the rev erse engineer is seeking, initially, to retain the code for calculating mortgage interest, while discarding the remainder of the program. A natural step would be to identify the code which implements mortgage calculations. Unfortunately pure slicing cannot help unless the engineer kno ws which *variables* are important for this computation. The engineer may be only partially familiar with the code and, therefore, unable to select a suitable variable or set of variables. Concept assignment can be used to produce a set of con tiguousstatements for which there is evidence that the code performs actions relating to mortgage interest, but the engineer cannot simply extract and reuse this code, since the code sequence is not an executable sub-program. How ever, by forming the ECS for the `Calculate:Mortgage In tere t` concept the rev erse engineer can extract the code of interest as a executable sub-program.

Selecting the 'calculate mortgage interest' concept produces the concept highlighted by light shading in the left-hand column of Figure 8. Figure 1 depicts the fragment of the domain model used to locate this concept. Using the algorithm in Figure 3 the ECS for calculate mortgage interest additionally identifies the boxed lines shown in the figure. Notice that the line of code

```
MOVE '010' TO APS-RECORD-IN.
```

6

Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)
1095-1350/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Brunel University. Downloaded on June 8, 2009 at 08:18 from IEEE Xplore. Restrictions apply.

Image 3 - Page Dilation Example: Input Image

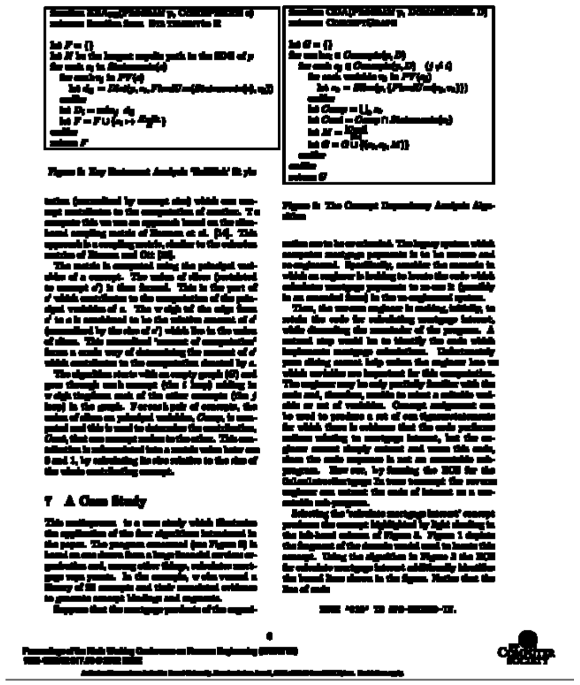


Image 4 - Page Dilation Example: Dilated image with a 4x4 kernel

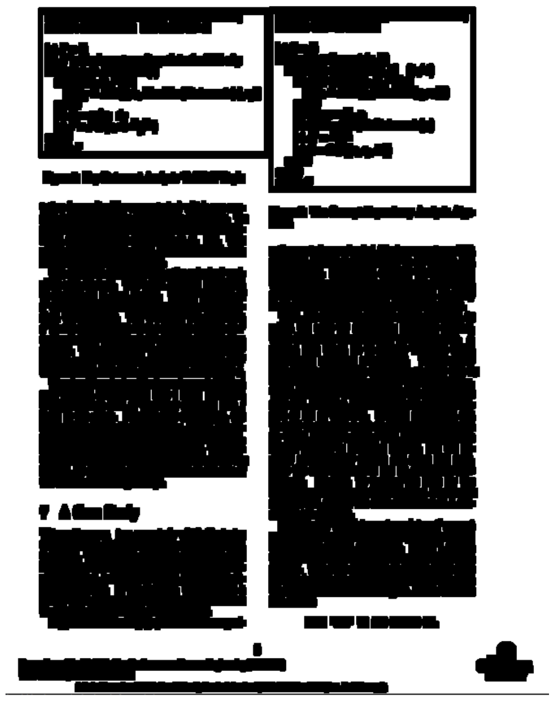
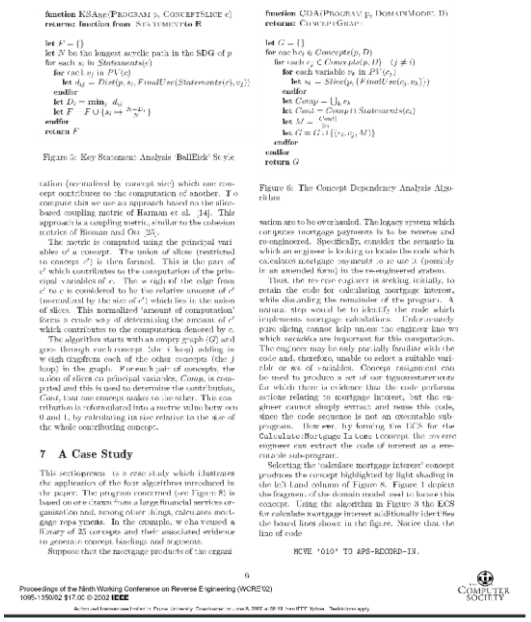


Image 5 - Page Dilation Example: Dilated image with an 8x11 kernel



```
function KSA_BE(PROGRAM p, CONCEPTSLICE c)
returns: function from STATEMENT to R

let $F = \{\}$
let $N$ be the longest acyclic path in the SDG of p
for each $s_i$ in $Statements(c)$
    for each $v_j$ in $PV(c)$
        let $d_{ij} = Dist(p, s_i, FinalUse(Statements(c), v_j))$
    endfor
    let $D_i = \min_j d_{ij}$
    let $F = F \cup \{s_i \mapsto \frac{N-D_i}{N}\}$
endfor
return $F$
```

Figure 5: Key Statement Analysis 'BallEick' Style

tation (normalised by concept size) which one concept contributes to the computation of another. To compute this we use an approach based on the slice-based coupling metric of Harman et al. [14]. This approach is a coupling metric, similar to the cohesion metrics of Bieman and Ott [25].

The metric is computed using the principal variables of a concept. The union of slices (restricted to concept c') is then formed. This is the part of c' which contributes to the computation of the principal variables of c. The weight of the edge from c' to c is considered to be the relative amount of c' (normalized by the size of c') which lies in the union of slices. This normalized 'amount of computation' forms a crude way of determining the amount of c' which contributes to the computation denoted by c.

The algorithm starts with an empty graph (G) and goes through each concept (the i loop) adding in weightingfrom each of the other concepts (the j loop) in the graph. For each pair of concepts, the union of slices on principal variables, $Comp$, is computed and this is used to determine the contribution, $Cont$, that one concept makes to the other. This contribution is reformulated into a metric value between 0 and 1, by calculating its size relative to the size of the whole contributing concept.

7 A Case Study

This sectiopresents a case study which illustrates the application of the four algorithms introduced in the paper. The program concerned (see Figure 8) is based on one drawn from a large financial services organisation and, among other things, calculates mortgage repayments. In the example, we have used a library of 25 concepts and their associated evidence to generate concept bindings and segments.

Suppose that the mortgage products of the organi

```
function CDA(PROGRAM p, DOMAINMODEL D)
returns: CONCEPTGRAPH

let $G = \{\}$
for each $c_i \in Concepts(p, D)$
    for each $c_j \in Concepts(p, D)$   $(j \neq i)$
        for each variable $v_k$ in $PV(c_j)$
            let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
        endfor
        let $Comp = \bigcup_k s_k$
        let $Cont = Comp \cap Statements(c_i)$
        let $M = \frac{|Cont|}{|c_i|}$
        let $G = G \cup \{(c_i, c_j, M)\}$
    endfor
endfor
return $G$
```

Figure 6: The Concept Dependency Analysis Algorithm

sation are to be overhauled. The legacy system which computes mortgage payments is to be reverse and re-engineered. Specifically, consider the scenario in which an engineer is looking to locate the code which calculates mortgage payments to re use it (possibly in an amended form) in the re-engineered system.

Thus, the reverse engineer is seeking, initially, to retain the code for calculating mortgage interest, while discarding the remainder of the program. A natural step would be to identify the code which implements mortgage calculations. Unfortunately pure slicing cannot help unless the engineer knows which *variables* are important for this computation. The engineer may be only partially familiar with the code and, therefore, unable to select a suitable variable or set of variables. Concept assignment can be used to produce a set of contiguous statements for which there is evidence that the code performs actions relating to mortgage interest, but the engineer cannot simply extract and reuse this code, since the code sequence is not an executable sub-program. However, by forming the ECS for the Calculate:Mortgage Interest concept the reverse engineer can extract the code of interest as a executable sub-program.

Selecting the 'calculate mortgage interest' concept produces the concept highlighted by light shading in the left hand column of Figure 8. Figure 1 depicts the fragment of the domain model used to locate this concept. Using the algorithm in Figure 3 the ECS for calculate mortgage interest additionally identifies the boxed lines shown in the figure. Notice that the line of code

```
MOVE '010' TO APS-RECORD-IN.
```

6

Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)
1095-1350/02 $17.00 © 2002 IEEE

COMPUTER SOCIETY

Image 6 - Page Dilation Example: Input image after frame removal

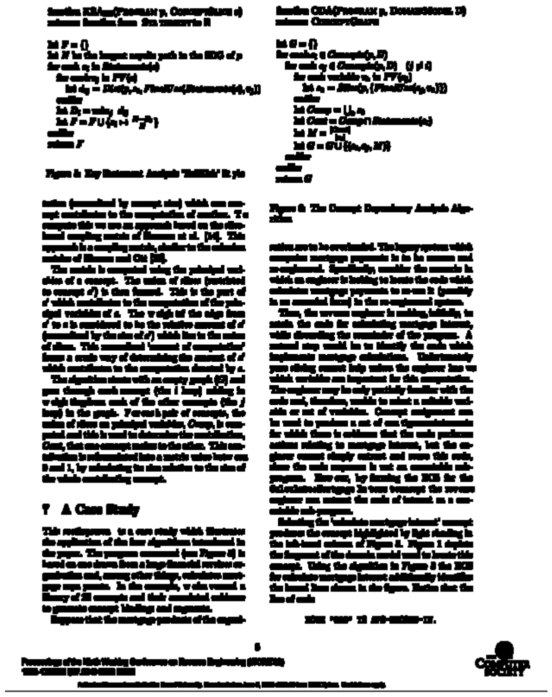


Image 7 - Page Dilation Example: Dilated image with a 4x4 kernel after frame removal

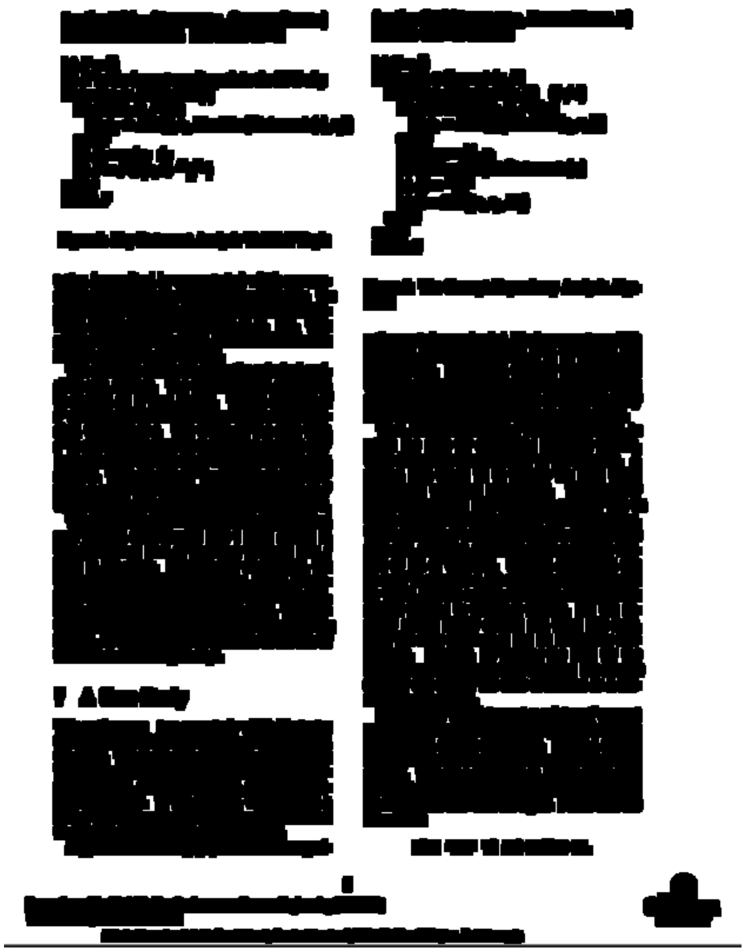


Image 8 - Page Dilation Example: Dilated image with a 4x4 kernel after frame removal

Note that, based on the IEEE Standards that are followed in the published scientific documents we process and adjust the kernel of the dilation process accordingly, so that we detect text paragraphs as text blocks rather than word blocks (in this case, kernel size would have to be smaller).

5.2.3 Decomposition and classification of the pseudo code sections

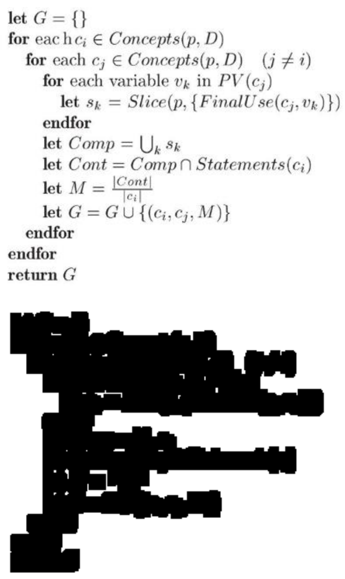
Following the pre-processing of an input image representing a document page, we have now obtained a set of rectangular blocks, in form of images, with each block representing a distinct element of the document. The main goal is to examine each of these blocks, using image processing techniques, and make decisions on whether they include algorithmic components. After a series of examples, we have identified four attributes which may indicate the given image describes algorithmic component. These four attributes are the text structural definition represented by the indents in the text, the number of specific keywords found in the text, the occurrence of pairs of certain keywords which may indicate loops, and percentage of the image area occupied by text.

5.2.4 Attributes used in the decision making

For each extracted block, four different features are evaluated for the identification of the block component as algorithmic: 1) the number of the indents in every line of the text appearing in the block, 2) the number of the keywords detected in the text of the block, 3) the aligned pairs of specific keywords that indicates beginning and end of loops or selection branches, and 4) the amount of the area in the block which is not occupied by text.

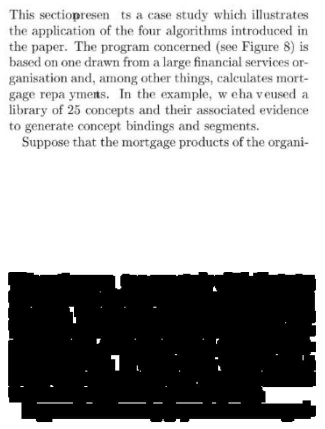
1. Indents in text

Algorithms are usually distinguished by human eye easily because of the structure of the text forming it, which is something that differentiates algorithms from plain text in paragraphs of technical documents. It is ordinary for a text paragraph that almost all of the text lines in it to begin at the leftmost point of the line, with the exception of the first line which formally includes an indent. In contrast to plain text components, algorithmic components, in a great number of cases, use indents at the very beginning of the lines to signify, along with the keywords, the levels of the execution and make the set of the algorithm commands more readable and maintainable. For the system to detect the number of the indents in the text of the block, the extracted text block is again dilated over a new kernel, which is now able to detect the structure of the text according to the higher size of the font in the text of the image. Following this and starting from the leftmost and topmost point of the image, when an indent is detected, it is considered the first level of nesting of the command component.



```
let $G = \{\}$
for eac h $c_i \in Concepts(p, D)$
  for each $c_j \in Concepts(p, D) \quad (j \neq i)$
    for each variable $v_k$ in $PV(c_j)$
      let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
    endfor
    let $Comp = \bigcup_k s_k$
    let $Cont = Comp \cap Statements(c_i)$
    let $M = \frac{|Cont|}{|c_i|}$
    let $G = G \cup \{(c_i, c_j, M)\}$
  endfor
endfor
return $G$
```

Image 9 - Text structure shape indicates algorithm

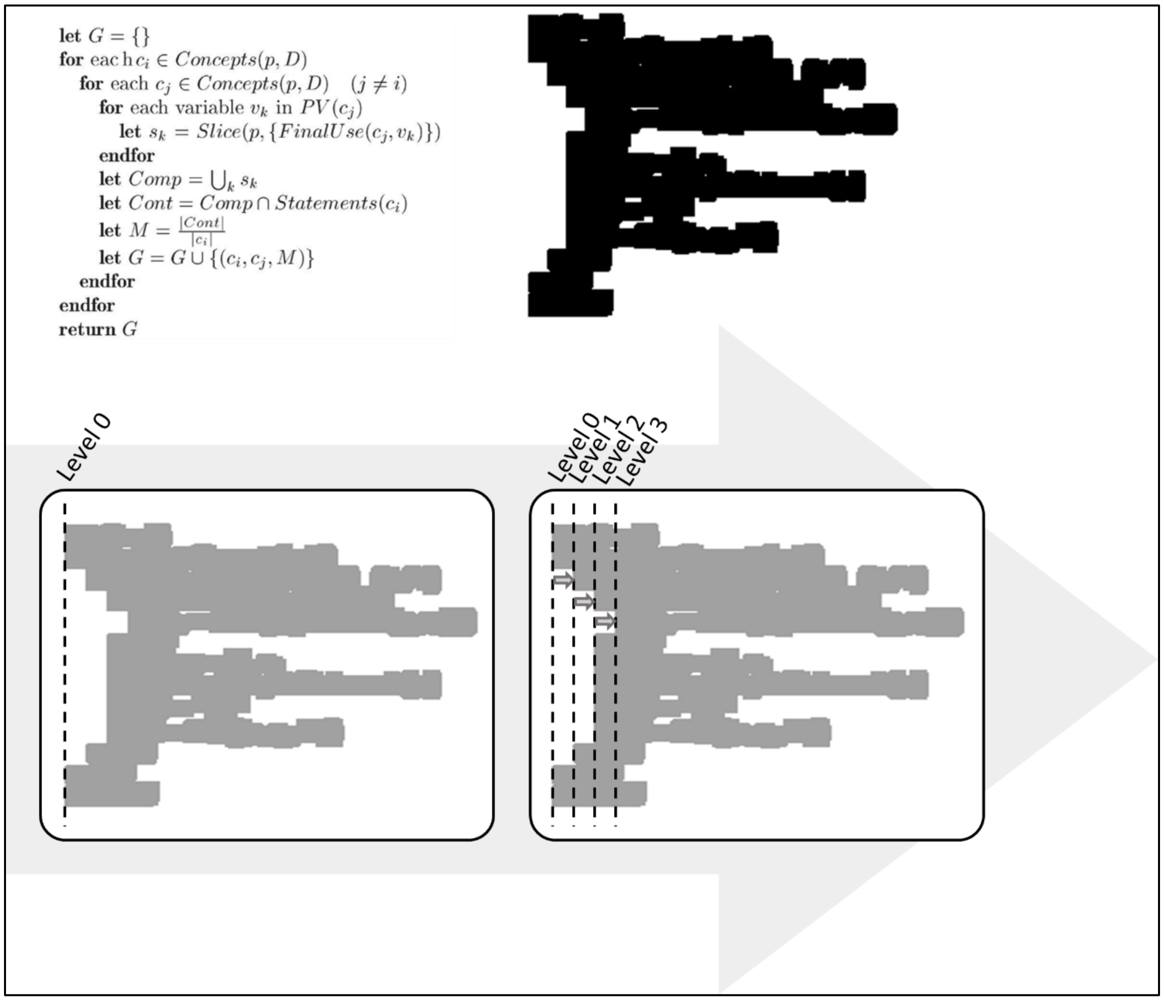


This sectiopresen ts a case study which illustrates the application of the four algorithms introduced in the paper. The program concerned (see Figure 8) is based on one drawn from a large financial services organisation and, among other things, calculates mortgage repa ymets. In the example, w eha veused a library of 25 concepts and their associated evidence to generate concept bindings and segments.
Suppose that the mortgage products of the organi-

Image 10 - Text structure shape indicates plain text in paragraph form

Accordingly, for each subsequent indent occurred, which will start towards to the right side of the previously detected indent (this is the case where the x coordinate of the detected pixel on the image will be higher than the x coordinate of the last level of nesting), the level of the nesting will be increasing by one. Finally, the number of the maximum level of nesting in the text will be kept as the feature describing the indents of the text in the image block.

Usually, blocks with maximum number of indents equal to one will not be considered to be algorithms, while blocks with a maximum number of indents being higher of one will have a higher probability to represent an algorithm.



```
let $G = \{\}$
for each $c_i \in Concepts(p, D)$
  for each $c_j \in Concepts(p, D)$   $(j \neq i)$
    for each variable $v_k$ in $PV(c_j)$
      let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
    endfor
    let $Comp = \bigcup_k s_k$
    let $Cont = Comp \cap Statements(c_i)$
    let $M = \frac{|Cont|}{|c_i|}$
    let $G = G \cup \{(c_i, c_j, M)\}$
  endfor
endfor
return $G$
```

Figure 14 - In the images above we see the extracted block after the intensity inversion where the white areas represent the text detected in the block while the black areas represent empty areas. In the left one we distinct the first level of loop nesting while in the one on the right we have proceeded gradually to all three levels of loop nesting.

2. Keywords

Another feature that contributes to the identification of a text as an algorithm, is the number of keywords that may be found in it. Each different programming operation is identified by a keyword, which is universal, especially for the case of pseudo-algorithms, where not a specific programming language is used, thus there are no limitations on the syntax. For example, loops are usually described through the keywords 'while' and 'for', which describe the 'while loop' and 'for loop', respectively. In the same context, there are keywords like 'if', 'then', 'function', 'end', 'input', 'output', etc. that may also describe algorithmic operations, but are also very likely to appear in plain text components. Finally, the keyword 'return' is largely found in function declarations or other algorithm components and can be used in the detection of an algorithm. For the extraction of the keywords in each block, we applied OCR methods where all the words of each text line were recognized.

3. Aligned pairs of keywords

Following the detection of specific keywords that are very often found in algorithms and describe specific algorithmic operations, we move one step forward and we aim to recognize pairs of keywords which are strongly connected to each other, and which are expected to be found one below the other in the algorithm. These sets of keywords usually indicate the start and end of a loop or a branch, or even the opening and closing of a method e.g. {'for','endfor'}, {'if','else'}, {'else','endif'} etc.

For the detection of this kind of sets, following the detection of the keywords which are to be examined, the position on the image of these keywords is also kept and compared to the rest of the keywords, towards the y coordinate of them. We have also set a small threshold for the deviation of these words in the x axis, for error purposes.

4. Amount of text occupied in image block

It is very ordinary for a text paragraph the text component to dominate towards the empty space in a text image. In order to have a measure for the amount that the lines of the text occupy in the mage, we will take under consideration the percentage of the x coordinates for which an imaginative line $x=x_0$ is more than 80% full of “black” (text). Following this process, in cases of text paragraphs, this percentage is going to be higher than 90%, while in algorithms, it will hardly exceed 50%. The following examples will give a better idea of the concept.

Two images in original and dilated form. On the upper one, the percentage of columns of pixels that are more than 80% full of text is 93, while on the lower example we see this percentage being 40%, giving a greater chance to describe an algorithm. Finally, after taking under consideration the evaluation of these four attributes, our system gives out the decision of whether the block contains an algorithm or not. This is a rule based approach as we have set specific values as thresholds for the classification process.

5.2.5 Learning

As a next step on the recognition of algorithms in published technical documents, we used a machine learning technique, in addition to our previously introduced rule-based approach, in order to improve the accuracy of the detection of the algorithms-pseudocodes from an input document image. For the prediction process, a logistic regression model was used, which is one of the simplest and commonly used Machine Learning algorithms [39]. This is a useful classification method mainly for solving a binary classification problem such ours, where we get to decide whether an input image is a description of an algorithm or not. Logistic regression is a statistical method for predicting binary classes. The outcome or target variable is dichotomous in nature.

After the dilation of the input image and the extraction of the several text blocks in the image, a logistic regression model is built, which based on specific features, will predict whether a text block contains an algorithmic component or not. The features used for the model to be trained and tested are the four attributes used in the rule-based classification process: the number of the indents in the text block, the number of keywords found, the count of the aligned pairs of keywords and the percentage of the text in the block image area. In this approach we included an extra feature representing the number of the pairs of the keywords regardless of their placement in the text, in order to check for unstructured text of algorithmic nature.

Logistic Regression is a special case of linear regression where the target variable is categorical in nature. It uses a log of odds as the dependent variable. Logistic Regression predicts the probability of occurrence of a binary event utilizing a logit function.

$$Y_i = f(X_i, \beta)$$

The equation of the linear regression, where Y_i represents the dependent variable and X_i represents the independent variable, f stands for the function, and β stands for the unknown parameters.

$$p = \frac{1}{1 + e^{-y}}$$

After the application of the Sigmoid function on linear regression ($p=1/1+e^{-y}$), this is the logistic regression function which describes our model.

Properties of Logistic Regression:

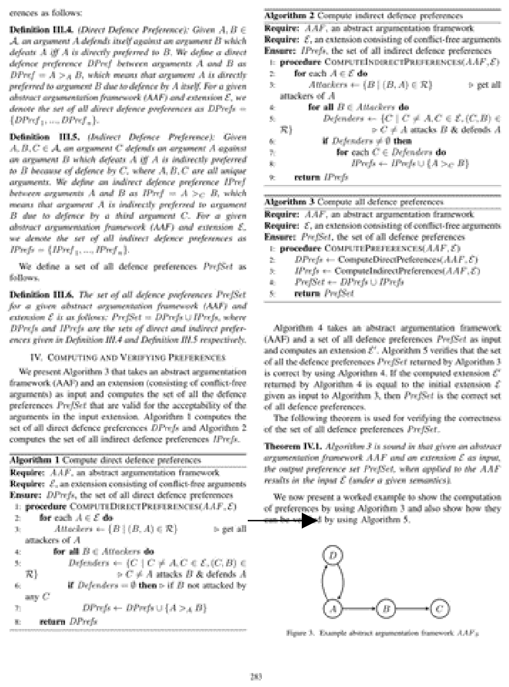
- The dependent variable in logistic regression follows Bernoulli Distribution.
- Estimation is done through maximum likelihood.

- No R Square, Model fitness is calculated through Concordance, KS-Statistics.

The Dataset

A number of 213 images were collected and processed to form our dataset. Each of the image input was an image representing a pdf page from a technical document, mainly in a two-column format. Each of the input images was preprocessed and gone through the pyramidal process in order to produce the several blocks which would be evaluated for the detection of algorithmic component. After the preprocessing, these collected pdf images in a .jpg format would give almost 3000 block elements for our dataset.

As an example of how the input images will be split into several blocks, the table below describes the several image blocks extracted from the following image:



erences as follows:

Definition III.4. *(Direct Defence Preference): Given $A, B \in \mathcal{A}$, an argument A defends itself against an argument B which defeats A iff A is directly preferred to B. We define a direct defence preference DPref between arguments A and B as $DPref = A >_A B$, which means that argument A is directly preferred to argument B due to defence by A itself. For a given abstract argumentation framework (AAF) and extension $\mathcal{E}$, we denote the set of all direct defence preferences as $DPrefs = \{DPref_1, ..., DPref_n\}$.*

Definition III.5. *(Indirect Defence Preference): Given $A, B, C \in \mathcal{A}$, an argument C defends an argument A against an argument B which defeats A iff A is indirectly preferred to B because of defence by C, where A, B, C are all unique arguments. We define an indirect defence preference IPref between arguments A and B as $IPref = A >_C B$, which means that argument A is indirectly preferred to argument B due to defence by a third argument C. For a given abstract argumentation framework (AAF) and extension $\mathcal{E}$, we denote the set of all indirect defence preferences as $IPrefs = \{IPref_1, ..., IPref_n\}$.*

We define a set of all defence preferences *PrefSet* as follows.

Definition III.6. *The set of all defence preferences PrefSet for a given abstract argumentation framework (AAF) and extension $\mathcal{E}$ is as follows: $PrefSet = DPrefs \cup IPrefs$, where DPrefs and IPrefs are the sets of direct and indirect preferences given in Definition III.4 and Definition III.5 respectively.*

IV. COMPUTING AND VERIFYING PREFERENCES

We present Algorithm 3 that takes an abstract argumentation framework (AAF) and an extension (consisting of conflict-free arguments) as input and computes the set of all the defence preferences *PrefSet* that are valid for the acceptability of the arguments in the input extension. Algorithm 1 computes the set of all direct defence preferences *DPrefs* and Algorithm 2 computes the set of all indirect defence preferences *IPrefs*.

```
Algorithm 1 Compute direct defence preferences
Require: $AAF$, an abstract argumentation framework
Require: $\mathcal{E}$, an extension consisting of conflict-free arguments
Ensure: $DPrefs$, the set of all direct defence preferences
1: procedure COMPUTEDIRECTPREFERENCES($AAF, \mathcal{E}$)
2:     for each $A \in \mathcal{E}$ do
3:         $Attackers \leftarrow \{B \mid (B, A) \in \mathcal{R}\}$        ▷ get all attackers of $A$
4:         for all $B \in Attackers$ do
5:             $Defenders \leftarrow \{C \mid C \neq A, C \in \mathcal{E}, (C, B) \in \mathcal{R}\}$    ▷ $C \neq A$ attacks $B$ & defends $A$
6:             if $Defenders = \emptyset$ then ▷ if $B$ not attacked by any $C$
7:                 $DPrefs \leftarrow DPrefs \cup \{A >_A B\}$
8:     return $DPrefs$
```

```
Algorithm 2 Compute indirect defence preferences
Require: $AAF$, an abstract argumentation framework
Require: $\mathcal{E}$, an extension consisting of conflict-free arguments
Ensure: $IPrefs$, the set of all indirect defence preferences
1: procedure COMPUTEINDIRECTPREFERENCES($AAF, \mathcal{E}$)
2:     for each $A \in \mathcal{E}$ do
3:         $Attackers \leftarrow \{B \mid (B, A) \in \mathcal{R}\}$        ▷ get all attackers of $A$
4:         for all $B \in Attackers$ do
5:             $Defenders \leftarrow \{C \mid C \neq A, C \in \mathcal{E}, (C, B) \in \mathcal{R}\}$    ▷ $C \neq A$ attacks $B$ & defends $A$
6:             if $Defenders \neq \emptyset$ then
7:                 for each $C \in Defenders$ do
8:                     $IPrefs \leftarrow IPrefs \cup \{A >_C B\}$
9:     return $IPrefs$
```

```
Algorithm 3 Compute all defence preferences
Require: $AAF$, an abstract argumentation framework
Require: $\mathcal{E}$, an extension consisting of conflict-free arguments
Ensure: $PrefSet$, the set of all defence preferences
1: procedure COMPUTEPREFERENCES($AAF, \mathcal{E}$)
2:     $DPrefs \leftarrow$ ComputeDirectPreferences($AAF, \mathcal{E}$)
3:     $IPrefs \leftarrow$ ComputeIndirectPreferences($AAF, \mathcal{E}$)
4:     $PrefSet \leftarrow DPrefs \cup IPrefs$
5:     return $PrefSet$
```

Algorithm 4 takes an abstract argumentation framework (AAF) and a set of all defence preferences *PrefSet* as input and computes an extension $\mathcal{E}'$. Algorithm 5 verifies that the set of all the defence preferences *PrefSet* returned by Algorithm 3 is correct by using Algorithm 4. If the computed extension $\mathcal{E}'$ returned by Algorithm 4 is equal to the initial extension $\mathcal{E}$ given as input to Algorithm 3, then *PrefSet* is the correct set of all defence preferences.

The following theorem is used for verifying the correctness of the set of all defence preferences *PrefSet*.

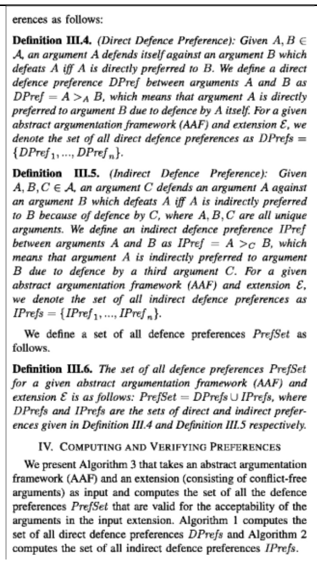
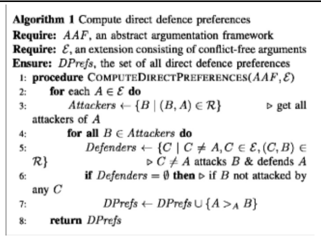
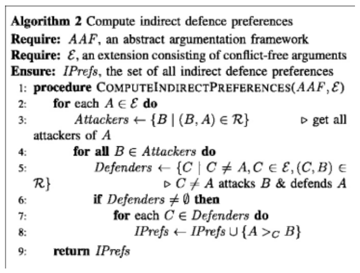
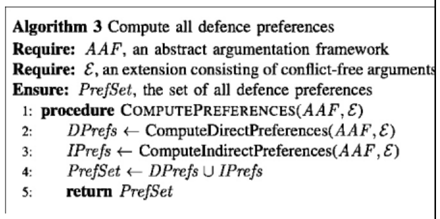
Theorem IV.1. *Algorithm 3 is sound in that given an abstract argumentation framework AAF and an extension $\mathcal{E}$ as input, the output preference set PrefSet, when applied to the AAF results in the input $\mathcal{E}$ (under a given semantics).*

We now present a worked example to show the computation of preferences by using Algorithm 3 and also show how they can be verified by using Algorithm 5.

Figure 3. Example abstract argumentation framework AAF_3

283

Image 11 – Example of input image

1	
2	
3	
4	

erences as follows:

Definition III.4. *(Direct Defence Preference): Given $A, B \in \mathcal{A}$, an argument A defends itself against an argument B which defeats A iff A is directly preferred to B. We define a direct defence preference DPref between arguments A and B as $DPref = A >_A B$, which means that argument A is directly preferred to argument B due to defence by A itself. For a given abstract argumentation framework (AAF) and extension $\mathcal{E}$, we denote the set of all direct defence preferences as $DPrefs = \{DPref_1, ..., DPref_n\}$.*

Definition III.5. *(Indirect Defence Preference): Given $A, B, C \in \mathcal{A}$, an argument C defends an argument A against an argument B which defeats A iff A is indirectly preferred to B because of defence by C, where A, B, C are all unique arguments. We define an indirect defence preference IPref between arguments A and B as $IPref = A >_C B$, which means that argument A is indirectly preferred to argument B due to defence by a third argument C. For a given abstract argumentation framework (AAF) and extension $\mathcal{E}$, we denote the set of all indirect defence preferences as $IPrefs = \{IPref_1, ..., IPref_n\}$.*

We define a set of all defence preferences *PrefSet* as follows.

Definition III.6. *The set of all defence preferences PrefSet for a given abstract argumentation framework (AAF) and extension $\mathcal{E}$ is as follows: $PrefSet = DPrefs \cup IPrefs$, where DPrefs and IPrefs are the sets of direct and indirect preferences given in Definition III.4 and Definition III.5 respectively.*

IV. COMPUTING AND VERIFYING PREFERENCES

We present Algorithm 3 that takes an abstract argumentation framework (AAF) and an extension (consisting of conflict-free arguments) as input and computes the set of all the defence preferences *PrefSet* that are valid for the acceptability of the arguments in the input extension. Algorithm 1 computes the set of all direct defence preferences *DPrefs* and Algorithm 2 computes the set of all indirect defence preferences *IPrefs*.

```
Algorithm 1 Compute direct defence preferences
Require: $AAF$, an abstract argumentation framework
Require: $\mathcal{E}$, an extension consisting of conflict-free arguments
Ensure: $DPrefs$, the set of all direct defence preferences
1: procedure COMPUTEDIRECTPREFERENCES($AAF, \mathcal{E}$)
2:     for each $A \in \mathcal{E}$ do
3:         $Attackers \leftarrow \{B \mid (B, A) \in \mathcal{R}\}$        ▷ get all attackers of $A$
4:         for all $B \in Attackers$ do
5:             $Defenders \leftarrow \{C \mid C \neq A, C \in \mathcal{E}, (C, B) \in \mathcal{R}\}$    ▷ $C \neq A$ attacks $B$ & defends $A$
6:             if $Defenders = \emptyset$ then ▷ if $B$ not attacked by any $C$
7:                 $DPrefs \leftarrow DPrefs \cup \{A >_A B\}$
8:     return $DPrefs$
```

```
Algorithm 2 Compute indirect defence preferences
Require: $AAF$, an abstract argumentation framework
Require: $\mathcal{E}$, an extension consisting of conflict-free arguments
Ensure: $IPrefs$, the set of all indirect defence preferences
1: procedure COMPUTEINDIRECTPREFERENCES($AAF, \mathcal{E}$)
2:     for each $A \in \mathcal{E}$ do
3:         $Attackers \leftarrow \{B \mid (B, A) \in \mathcal{R}\}$        ▷ get all attackers of $A$
4:         for all $B \in Attackers$ do
5:             $Defenders \leftarrow \{C \mid C \neq A, C \in \mathcal{E}, (C, B) \in \mathcal{R}\}$    ▷ $C \neq A$ attacks $B$ & defends $A$
6:             if $Defenders \neq \emptyset$ then
7:                 for each $C \in Defenders$ do
8:                     $IPrefs \leftarrow IPrefs \cup \{A >_C B\}$
9:     return $IPrefs$
```

```
Algorithm 3 Compute all defence preferences
Require: $AAF$, an abstract argumentation framework
Require: $\mathcal{E}$, an extension consisting of conflict-free arguments
Ensure: $PrefSet$, the set of all defence preferences
1: procedure COMPUTEPREFERENCES($AAF, \mathcal{E}$)
2:     $DPrefs \leftarrow$ ComputeDirectPreferences($AAF, \mathcal{E}$)
3:     $IPrefs \leftarrow$ ComputeIndirectPreferences($AAF, \mathcal{E}$)
4:     $PrefSet \leftarrow DPrefs \cup IPrefs$
5:     return $PrefSet$
```

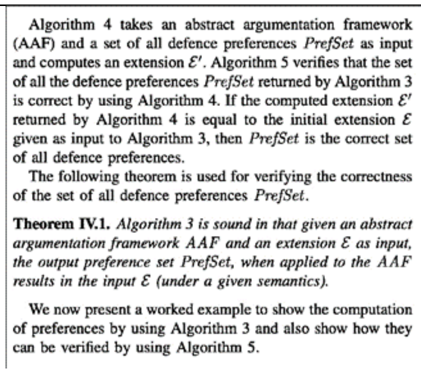
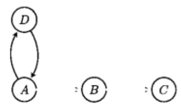
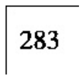
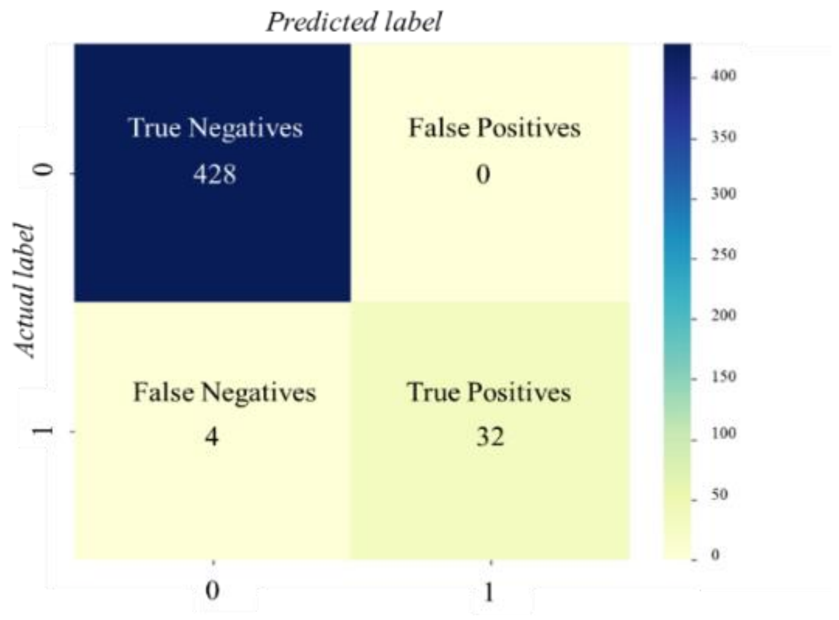
5	 Algorithm 4 takes an abstract argumentation framework (AAF) and a set of all defence preferences *PrefSet* as input and computes an extension $\mathcal{E}'$. Algorithm 5 verifies that the set of all the defence preferences *PrefSet* returned by Algorithm 3 is correct by using Algorithm 4. If the computed extension $\mathcal{E}'$ returned by Algorithm 4 is equal to the initial extension $\mathcal{E}$ given as input to Algorithm 3, then *PrefSet* is the correct set of all defence preferences. The following theorem is used for verifying the correctness of the set of all defence preferences *PrefSet*. **Theorem IV.1.** *Algorithm 3 is sound in that given an abstract argumentation framework AAF and an extension $\mathcal{E}$ as input, the output preference set PrefSet, when applied to the AAF results in the input $\mathcal{E}$ (under a given semantics).* We now present a worked example to show the computation of preferences by using Algorithm 3 and also show how they can be verified by using Algorithm 5.
6	 Figure 3. Example abstract argumentation framework AAF_3
7	 283
8	Authorized licensed use limited to: WRIGHT STATE UNIVERSITY. Downloaded on February 16,2021 at 16:58:29 UTC from IEEE X

Table 9 - The several image parts extracted from an image through dilation. Each individual image part is a new image in the dataset

Evaluation of learning process

For the evaluation process of the model performance and our features effectiveness, our dataset is divided into a training set and a test set. A number of experiments took place, with different ratios of data splitting, 70%-30%, and 80%-20% and 90%-10% following the Stratified cross validation technique.

The metrics of our model on the specific features returns an accuracy of 0.99, while the precision was the absolute 1 and the metric of recall equal to 0.88. Overall metric is the F1 score, which was calculated at 0.94, implying a satisfying model for our predictions. The metric of the area under the ROC curve is 0.997 which also verifies the model promising results.



Accuracy	0.991
Precision	1.0
Recall	0.888
F1	0.941
AUC	0.997

Image 12 - Output metrics of accuracy, precision, recall, F1 score and area under the curve.

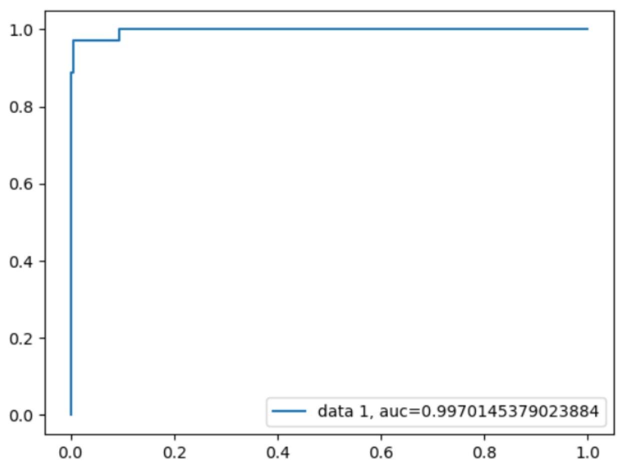


Figure 15 - Receiver Operating Characteristic(ROC) curve is a plot of the true positive rate against the false positive rate. It shows the tradeoff between sensitivity and specificity. AUC of 1 shows the ability to identify all true positives while avoiding false positives. Applying a learning process in our system is highly promising as the system will be able to recognize algorithmic components in the image which are not easily distinguished to be pseudo-algorithms.

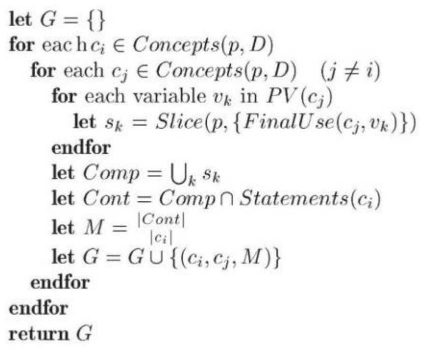
5.3 Translation of algorithms to graphs and SPN

5.3.1 Generation of graph

After the recognition of any image block as an algorithm, we move towards the analysis of the algorithmic component. For the part of the understanding of the algorithm, we aim to extract the pseudo-algorithm text and, based on the structure of it, generate a structural graph which represents the flow of the pseudocode steps execution. This is an automated procedure that will make use of previously used techniques which will now have different contribution to the system process.

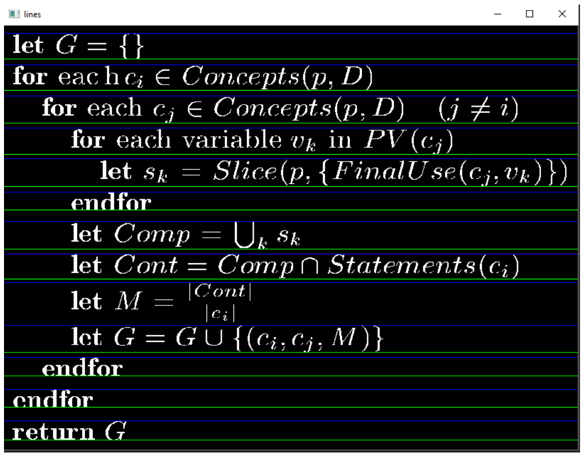
For the process of the translation of an image with algorithmic content into a structural graph, the image will first get analyzed in a structural level, where the distinct lines of text will be detected and the upper and lower bound of each line will be kept and visualized with blue and green color, respectively.

The name of the keyword initiating each line, along with the number of the nesting level will be the two factors which make the decision for the potential splitting of the algorithm in an additional branch. This results to some lines forming a branch by themselves, while other lines getting merged and encapsulated in a single branch.



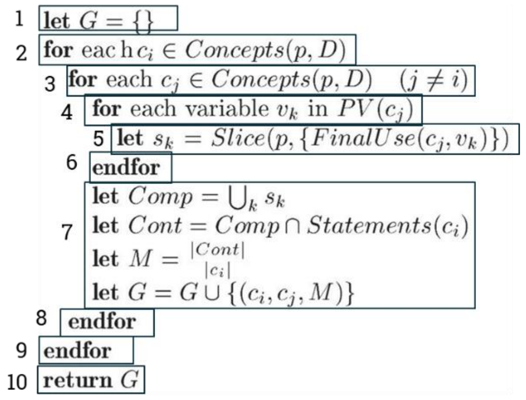
```
let $G = \{\}$
for eac h $c_i \in Concepts(p, D)$
   for each $c_j \in Concepts(p, D) \quad (j \neq i)$
      for each variable $v_k$ in $PV(c_j)$
         let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
      endfor
      let $Comp = \bigcup_k s_k$
      let $Cont = Comp \cap Statements(c_i)$
      let $M = \frac{|Cont|}{|c_i|}$
      let $G = G \cup \{(c_i, c_j, M)\}$
   endfor
endfor
return $G$
```

Image 13 - Input block with algorithmic components detected input to the process of graph representation



```
let $G = \{\}$
for eac h $c_i \in Concepts(p, D)$
   for each $c_j \in Concepts(p, D) \quad (j \neq i)$
      for each variable $v_k$ in $PV(c_j)$
         let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
      endfor
      let $Comp = \bigcup_k s_k$
      let $Cont = Comp \cap Statements(c_i)$
      let $M = \frac{|Cont|}{|c_i|}$
      let $G = G \cup \{(c_i, c_j, M)\}$
   endfor
endfor
return $G$
```

Image 14 - Detected text lines and their borders in input image block



```
1  let $G = \{\}$
2  for eac h $c_i \in Concepts(p, D)$
3     for each $c_j \in Concepts(p, D) \quad (j \neq i)$
4        for each variable $v_k$ in $PV(c_j)$
5           let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
6        endfor
         let $Comp = \bigcup_k s_k$
7        let $Cont = Comp \cap Statements(c_i)$
         let $M = \frac{|Cont|}{|c_i|}$
         let $G = G \cup \{(c_i, c_j, M)\}$
8     endfor
9  endfor
10 return $G$
```

Image 15 - Algorithm lines processed, assigned to branches, and some are grouped as a single branch

For every line, the first word detected in it is assigned to be the command-word, the keyword which represents the type of the command that is running in this specific line.

During the process of the analysis of the text component in a block of an algorithm, the placements of the command-word are evaluated for

- the decision of each line-command to be assigned to a new branch or an existing one, and
- the next line-command that will be executed if we described the steps of the algorithms with a flow chart diagram.

The process of assigning line-commands to branches and enumerating them, followed by the process of detecting the order of the line-commands during the execution process is presented in the next flow chart diagram.

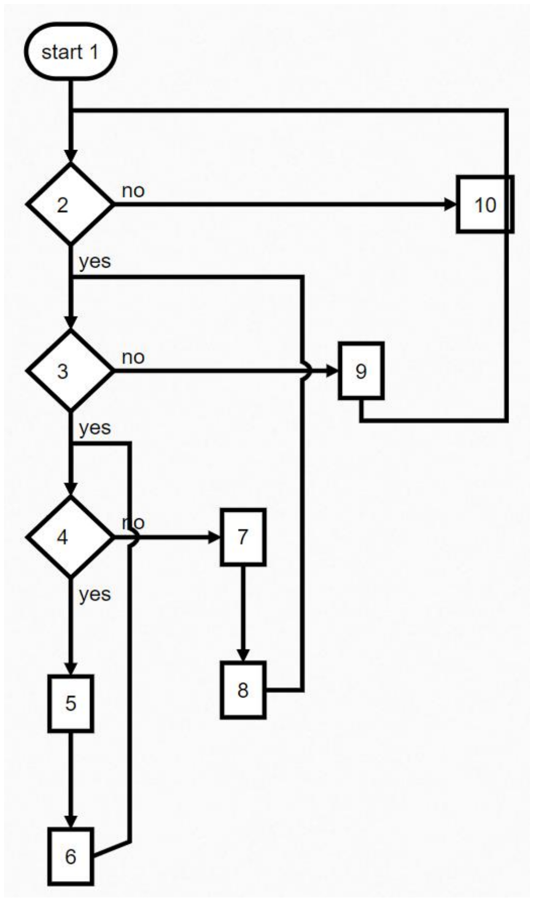


Figure 16 - Detected algorithm is translated to a graph representation in the form of a flow chart graph. Each number corresponds to a numbered pseudo-code block as presented above. Branches with line-commands where a condition is checked to be true or false in order the execution to move on accordingly, are represented with the diamond shapes. In this representation, evaluation of True is depicted with the 'yes', while False statements in conditions are described with 'no'. The rest of the branches are illustrated with rectangles.

5.3.2 Stochastic Petri Net Representation

The main purpose for this work is the translation of an image illustrating an algorithm into a Stochastic Petri Net representation, which is the final step of the system's process. This step takes place after the graph representation as it enhances the structural representation offered by a graph with the functional information (timing and synchronization) of the translated component.

Every line-command, in order to be executed needs to have the token which will cause the transition to fire. In cases of branches with conditions, the evaluation of the condition will give or not the token to the place representing the branch in order for it to be executed and move to the transition.

In our representation, the circles represent the places, and the rectangular shapes represent the transitions.

<u>Linear execution of commands</u>

For the commands of the algorithm that follow a linear execution, the sample SPN is presented below. In this example illustrated in the image below, the branches 7 and 8 will execute linearly, one after the other, and the transition will fire when the evaluation of the commands in branch 7 is complete.

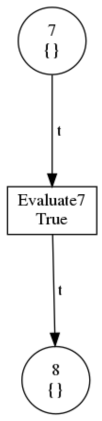


Figure 17 - Linear execution of command in SPN

Non-linear command execution

In cases of for loops, repetitions, branches such as if statements, steps, and switching cases, the execution of one line-command does not necessarily ensure the execution of the next in line branch. This is where the direction of the execution depends on the validity of the condition in the command-keyword of the branch. In the following example, we present the execution of a for loop, where the command lines nested in the loop will get executed only while the condition checked in branch is true. For this purpose, for the conditions detected after the command-keywords, a new place is created in order to check exclusively the veracity of the condition.

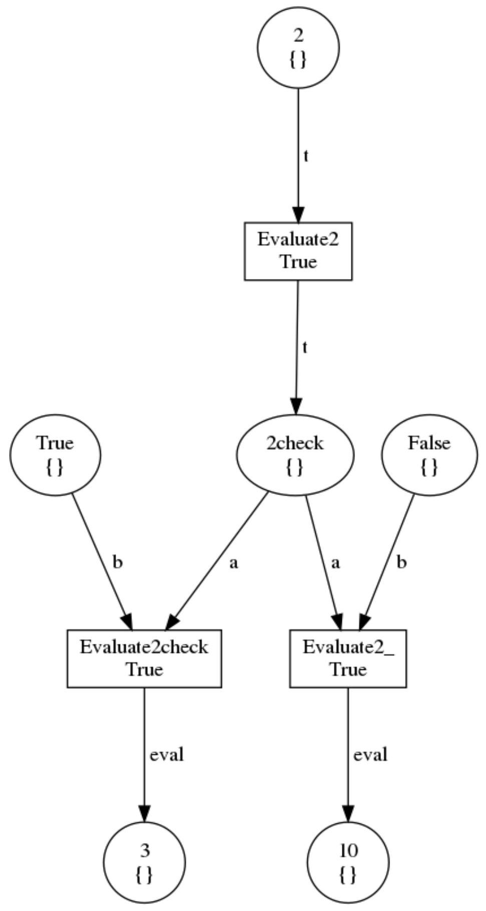
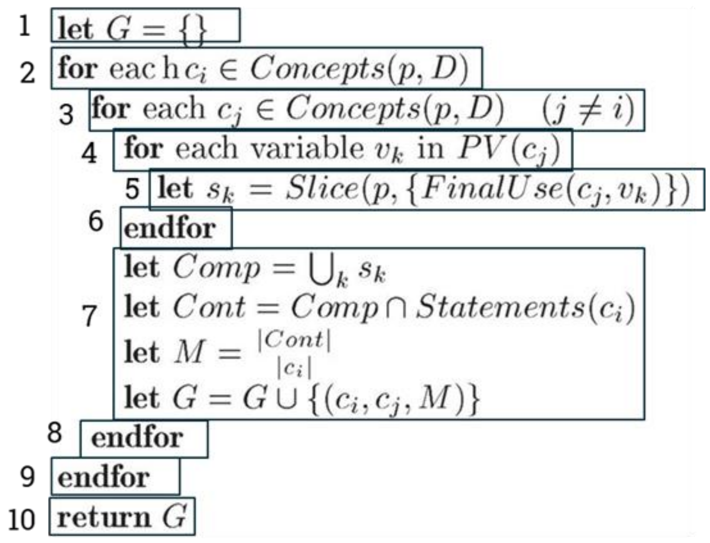


Figure 18 - Branch 2 represents a for loop and branch 2check contains the check of the value of the variable to be within limits in order for the execution of the loop to continue. While this check returns True, the Evaluate2check transition will get fired and the execution will proceed with branch 3. Otherwise, Evaluate2_ will fire and the execution will move forward with the commands in branch 10.

According to the two previously described examples, the generated SPN of the image translated to the flow chart diagram above is presented below. The execution begins at the place with the name Start and finishes at the place which does not point to any transaction - in our case this is place 10.



```
1  let $G = \{\}$
2  for eac h $c_i \in Concepts(p, D)$
 3  for each $c_j \in Concepts(p, D) \quad (j \neq i)$
  4  for each variable $v_k$ in $PV(c_j)$
   5  let $s_k = Slice(p, \{FinalUse(c_j, v_k)\})$
  6  endfor
     let $Comp = \bigcup_k s_k$
  7  let $Cont = Comp \cap Statements(c_i)$
     let $M = \frac{|Cont|}{|c_i|}$
     let $G = G \cup \{(c_i, c_j, M)\}$
 8  endfor
9  endfor
10 return $G$
```

Figure 19 - Segregation of commands or set of commands that belong in a loop

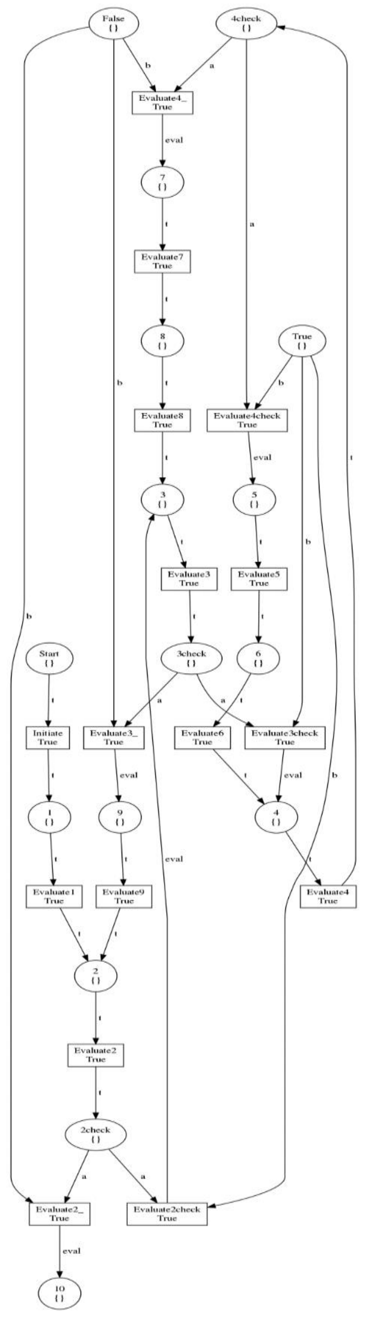


Figure 20 - SPN representation of a complete algorithmic component in a document

6. Conclusion and Future Work

In this thesis, we introduced a methodology to parse and recognizing mathematical formulas in LaTeX format. A rule-based methodology to parse and understand a mathematical formula is introduced and our purpose is to translate it to an SPN representation by first converting it to an attribute graph representation. As an additional functionality to our parsing methodology, we utilized a formal language to be able to create new randomly generated mathematical formulas as an input for parsing and also to provide syntactical evaluation to the detected formulas. As numerous research efforts in the field of mathematical formulas processing have taken place through times, our effort was focused on the parsing of mathematical formulas with the aim of converting them into an SPN state machine, which provides, not only structural information, but also functional information such as timing and synchronization.

As the main contribution of this work, we implemented a system to automatically parse technical documents in order to detect algorithmic component. The presented hybrid approach offers good satisfying results from the prediction model and which makes it a unique successful methodology as a methodology to handle algorithms from a computer vision perspective. In addition, the SPN representation of the detected algorithms enhances the understanding of the machine towards the analysis of technical documents containing pseudo-algorithms and at the same time offers a different approach towards its functional implementation.

For future work, further extensions of the document pseudo algorithm can be made by converting math formulas included in pseudo algorithms into SPNs and comparing the outcomes. For example, semantical analysis and interpretation of mathematical formulas can be studied and incorporated into the initial algorithm. Finally, additional applications of the algorithmic component analysis methodology can be explored.

REFERENCES

[1] N. Bourbakis, "Converting Diagrams, Formulas, Tables, Graphics and Pictures into SPN and NL-text Sentences for Automatic Deep Understanding of Technical Documents," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, Boston, MA, 2017.

[2] A. Angeleas, N. Bourbakis and G. Tsihrintzis, "Categorization of research surveys and reviews on human activities," in *2016 7th International Conference on Information, Intelligence, Systems & Applications (IISA)*, Chalkidiki, 2016.

[3] K.-f. Chan and D.-Y. Yeung, "Mathematical Expression Recognition: A Survey," *International Journal on Document Analysis and Recognition,* vol. 3, 2001.

[4] R. Zanibbi and D. Blostein, "Recognition and retrieval of mathematical expressions," *International Journal on Document Analysis and Recognition (IJDAR),* vol. 15, p. 331–357, 2012.

[5] S. Manganas and N. Bourbakis, "A Comparative Survey on Simultaneous EEG-fMRI Methodologies," in *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*, Washington, DC, 2017.

[6] R. Anderson, "Syntax-directed recognition of hand-printed two-dimensional mathematics," *Interactive Systems for Experimental Applied Mathematics,* pp. 436-459, 1968.

[7] K. Chan and D. Yeung, "Towards efficient structural analysis of mathematical expressions," in *Advances in Pattern Recognition. SSPR /SPR 1998. Lecture Notes in Computer Science*, 1998.

[8] K.-F. Chan and D.-Y. Yeung, "PenCalc: a novel application of on-line mathematical expression recognition technology," in *Proceedings of Sixth International Conference on Document Analysis and Recognition*, Seattle, WA, USA, 2001.

[9] J. Toumit, S. Garcia-Salicetti and H. Emptoz, "A hierarchical and recursive model of mathematical expressions for automatic reading of mathematical documents," in *Proceedings of the Fifth International Conference on Document Analysis and Recognition. ICDAR '99 (Cat. No.PR00318)*, Bangalore, India, 1999.

[10] Y. Chen, T. Shimizu and M. Okada, "Fundamental study on structural understanding of mathematical expressions," 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028), Tokyo, Japan, 1999.

[11] Y. Chen, T. Shimizu, K. Yamauchi and M. Okada, "Ambiguous problem investigation in off-line mathematical expression understanding," in *2000 IEEE International Conference on Systems, Man and Cybernetics.*

[12] J.-M. Jin, Z. Han and Q.-R. Wang, "Typeset mathematical expression analysis," in *Proceedings of International Conference on Machine Learning and Cybernetics*, Beijing, China, 2002.

[13] S. Toyota, S. Uchida and M. Suzuki, "Structural Analysis of Mathematical Formulae with Verification Based on Formula Description Grammar," *Document Analysis Systems VII. DAS 2006. Lecture Notes in Computer Science,* vol. 3872, 2006.

[14] S. Lavirotte and L. Pottier, "Mathematical formula recognition using graph grammar," in *Proc SPIE. 3305.*, 1998.

[15] B. Chaudhuri and U. Garain, "An Approach for Recognition and Interpretation of Mathematical Expressions in Printed Document," *Pattern Analysis & Applications,* no. 3, p. 120–131, 2000.

[16] U. Garain and B. Chaudhuri, "A syntactic approach for processing mathematical expressions in printed documents," in *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, Barcelona, Spain, 2000.

[17] Y. Guo, L. Huang, C. Liu and X. Jiang, "An Automatic Mathematical Expression Understanding System," in *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, Parana, 2007.

[18] R. Fateman and T. Tokuyasu, "Progress in recognizing typeset mathematics," San Jose, CA, 1996.

[19] R. Fateman, T. Tokuyasu, B. Berman and N. Mitchell, "Optical character recognition and parsing of typeset mathematics," *Journal of Visual Communication and Image Representation,* vol. 7, no. 1, pp. 2-15, 1996.

[20] R. Zanibbi, D. Blostein and J. R. Cordy, "Recognizing mathematical expressions using tree transformation," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 24, no. 11, pp. 1455-1467, 2002.

[21] "Lexical analysis - Wikipedia, the free encyclopedia," 19 Sept 2021. [Online]. Available: https://en.wikipedia.org/wiki/Lexical_analysis. [Accessed 19 Sept 2021].

[22] Y. Takiguchi, M. Okada and Y. Miyake, "A fundamental study of output translation from layout recognition and semantic understanding system for mathematical formulae," in *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, Seoul, South Korea, 2005.

[23] D. Knuth, "Estimating the Efficiency of Backtrack Programs," *Mathematics of Computation,* vol. 29, no. 129, pp. 121-136, Jan. 1975.

[24] N. Bourbakis and S. J. Mertoguno, "A Holistic Approach for Automatic Deep Understanding and Protection of Technical Documents," *Int. Journal on AI Tools,* vol. 29, 2020.

[25] E. E. Kostalia, E. G. M. Petrakis and N. Bourbakis, "Evaluating Methods for the Parsing and Understanding of Mathematical Formulas in Technical Documents," in *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2020.

[26] M. Mahdavi, M. Condon, K. Davila and R. Zanibbi, "LPGA: Line-of-Sight Parsing with Graph-Based Attention for Math Formula Recognition," 2019.

[27] Advances in Pattern Recognition: Joint IAPR International Workshops, SSPR'98 and SPR'98, Sydney, Australia, August 11-13, 1998, Proceedings.

[28] H.-J. Lee and M.-C. Lee, "Understanding mathematical expressions using procedure-oriented transformation," *Pattern Recognition,* vol. 27, no. 3, pp. 447-457, March 1994.

[29] "Petri net - Wikipedia, the free encyclopedia," 20 April 2021. [Online]. Available: https://en.wikipedia.org/wiki/Petri_net. [Accessed 20 April 2021].

[30] "Petri Nets," Online, 20 April 2021. [Online]. Available: http://www.techfak.unibielefeld.de/~mchen/BioPNML/Intro/pnfaq.html. [Accessed 20 April 2021].

[31] A. Psarologou and N. Bourbakis, "Glossa - A Formal Language as a Mapping Mechanism NL Sentences into SPN State Machine for Actions/Events Association," *International Journal on Artificial Intelligence Tools,* vol. 26, 2017.

[32] J. Wang, "Petri nets for dynamic event-driven system modeling," in *Handb. Dyn. Syst. Model.*, 2007, pp. 1-17.

[33] F. Pommereau, "SNAKES: A flexible high-level Petri nets library," in *International Conference on Applications and Theory of Petri Nets and Concurrency*, 2015.

[34] [Online]. Available: https://www.latex-project.org.

[35] N. Chomsky, "Three Models for the Description of Language," *IRE Transactions on Information Theory,* vol. 2, no. 3, pp. 113-124, 1956.

[36] A. Psarologou and N. Bourbakis, "Glossa: A Formal Language as a Mapping Mechanism NL Sentences into SPN State Machine for Actions/Events Association," *International Journal of Artificial Intelligence Tools,* vol. 26, no. 2, 2017.

[37] N. Bourbakis and A. Klinger, "Hierarchical Picture Coding Scheme," *Pattern Recognition,* vol. 22, no. 3, pp. 317-329, 1989.

[38] R. Keefer and N. Bourbakis, "Interaction with a Mobile Reader for the Visually Impaired," in *2009 21st IEEE International Conference on Tools with Artificial Intelligence*, Newark, NJ, 2009.

[39] J. Peng, K. Lee and G. Ingersoll, "An Introduction to Logistic Regression Analysis and Reporting," *Journal of Educational Research - J EDUC RES.,* pp. 3-14, 2002.

[40] R. Zanibbi and D. Blostein, "Recognition and retrieval of mathematical expressions," in *IJDAR 15*, 2012.